UNIVERSITY OF ALBERTA

Inter-Procedural Analysis

CMPUT 497/500
Foundations of Program Analysis

Karim Ali
@karimhamdanali

Previously

- Points-to
- Aliases
- Must and May analyses
- Incomplete Programs
- Weak vs Strong Updates
- Access Paths
- Distributivity

Inter-Procedural Data-Flow Analysis

- Beyond procedure boundaries
  - Model the effects of
    - calls in the callers, and
    - calling contexts in the callees

Inter-Procedural Data-Flow Analysis

- Approaches
  - Generic: Call-strings approach, functional approach
  - Problem specific: Alias analysis, Points-to analysis, Partial redundancy elimination, Constant propagation

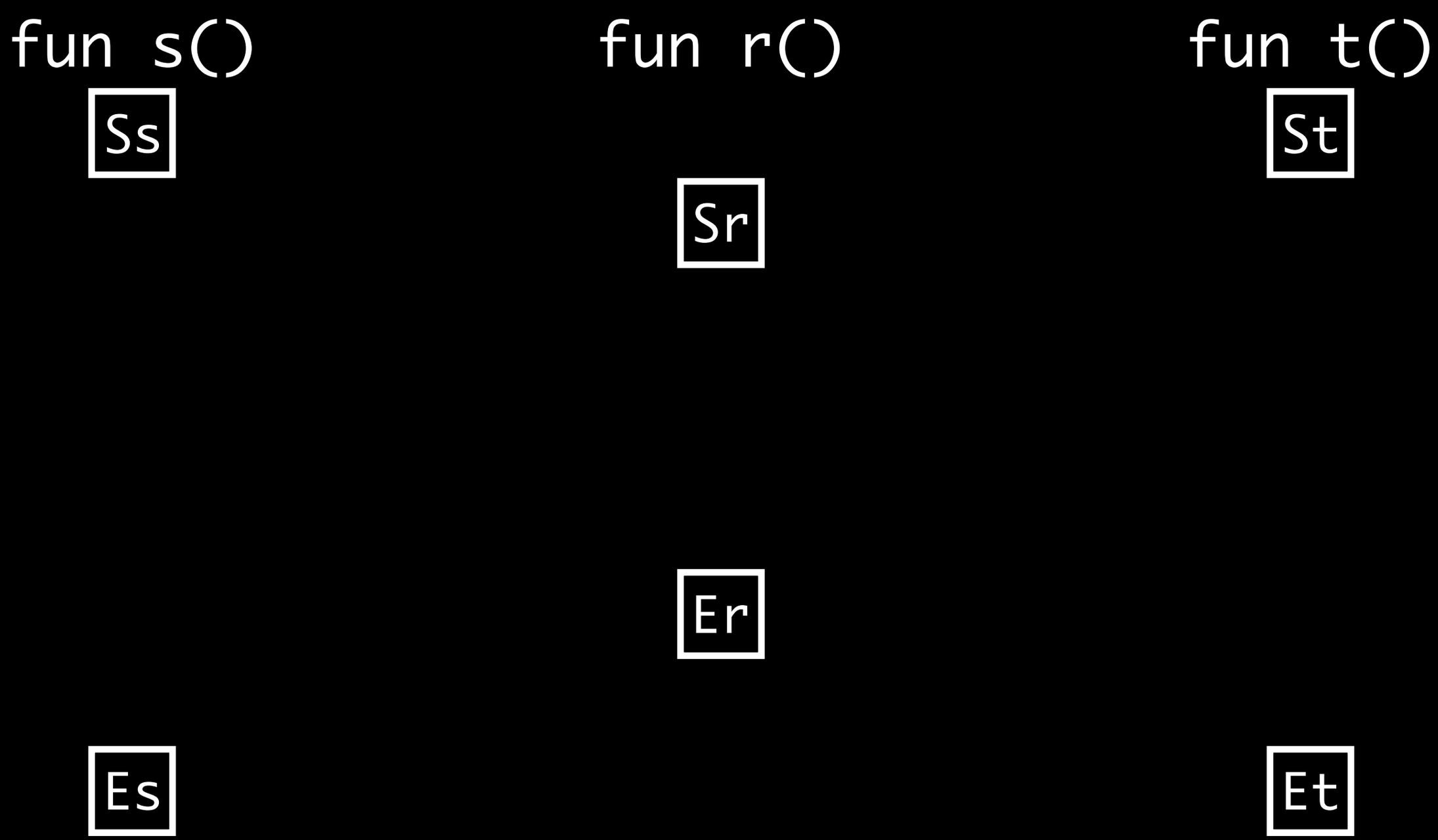
Inter-Procedural Data-Flow Analysis

```
fun s()
```

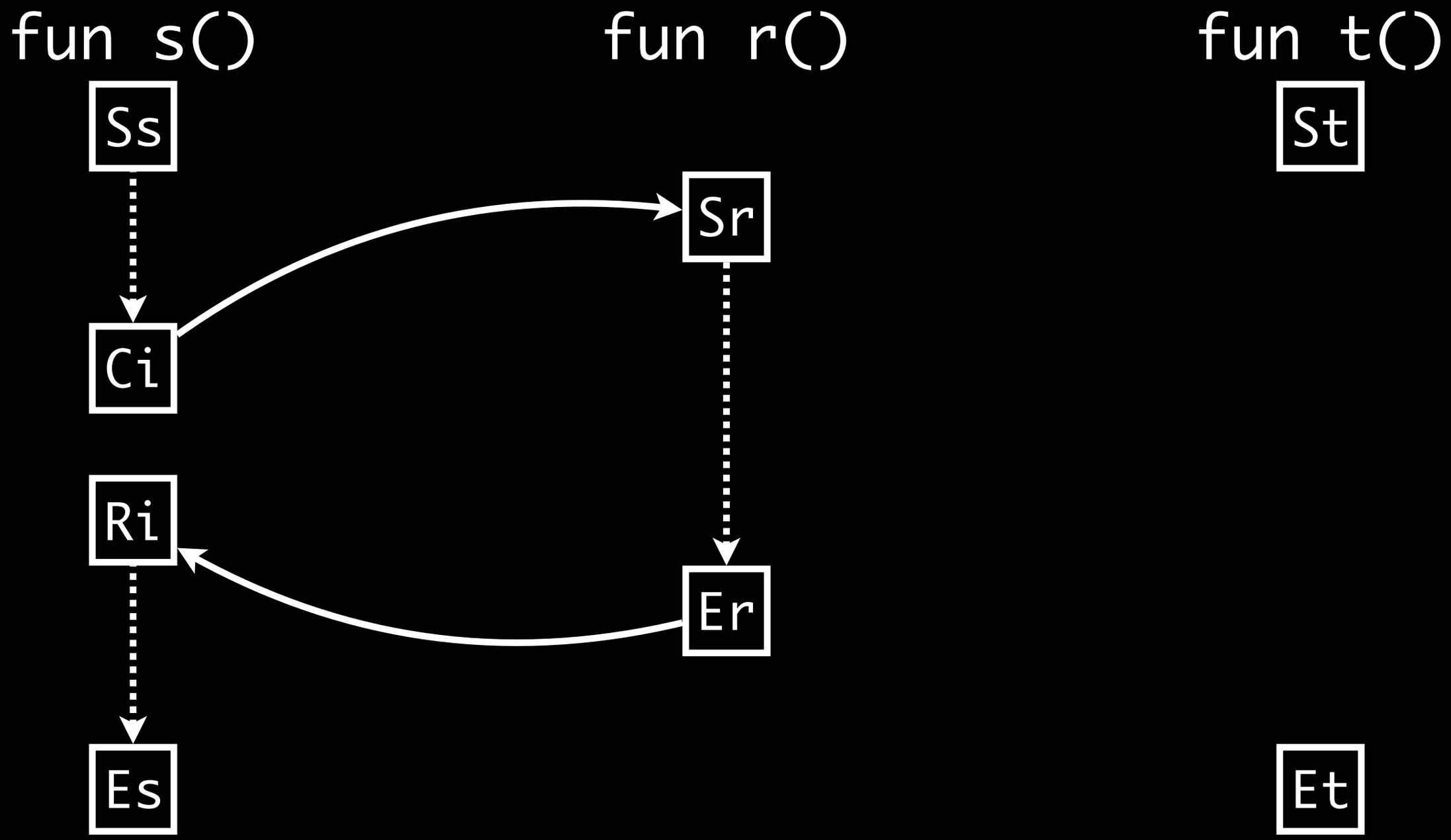
```
fun r()
```

```
fun t()
```

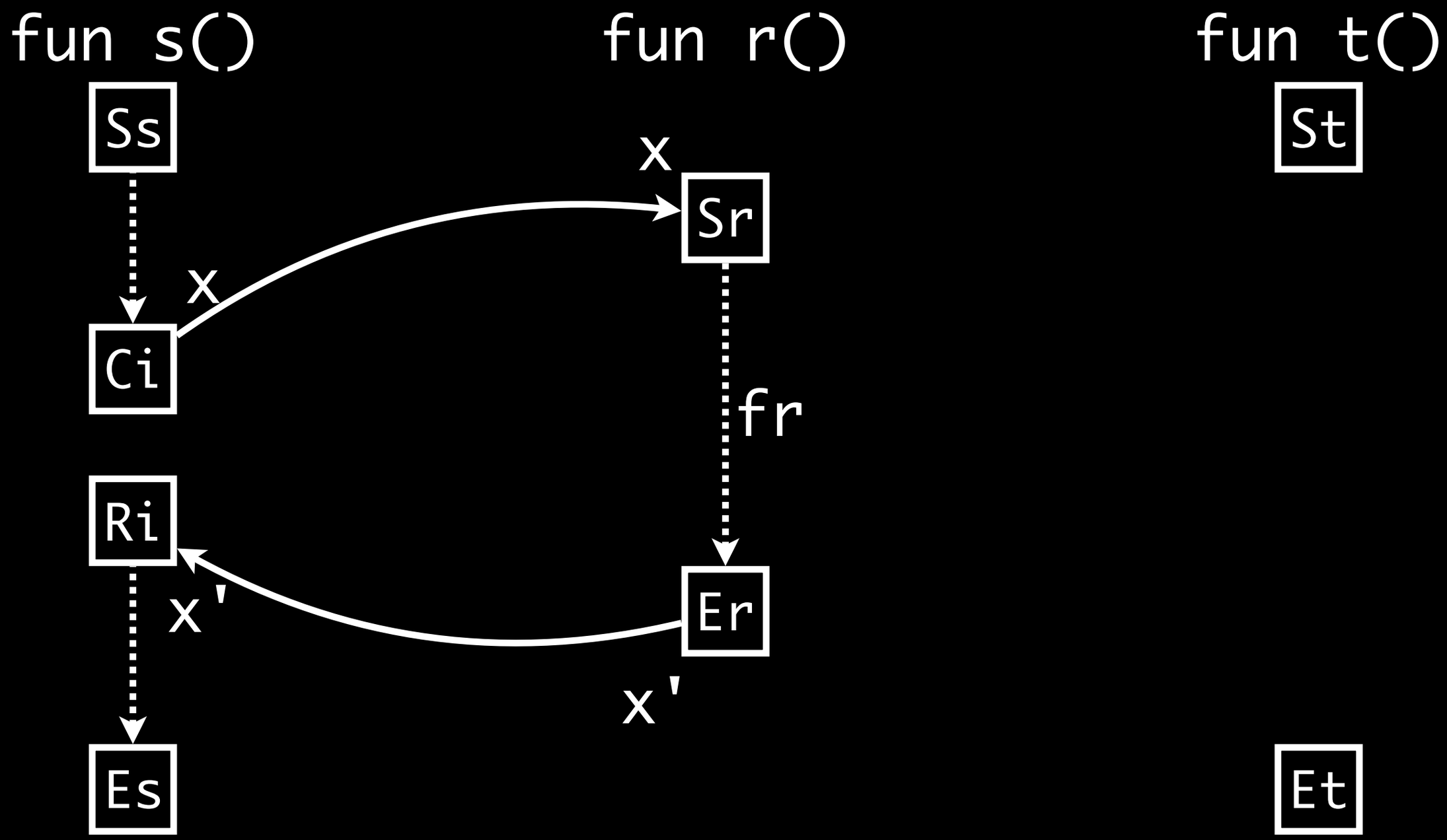
Inter-Procedural Data-Flow Analysis



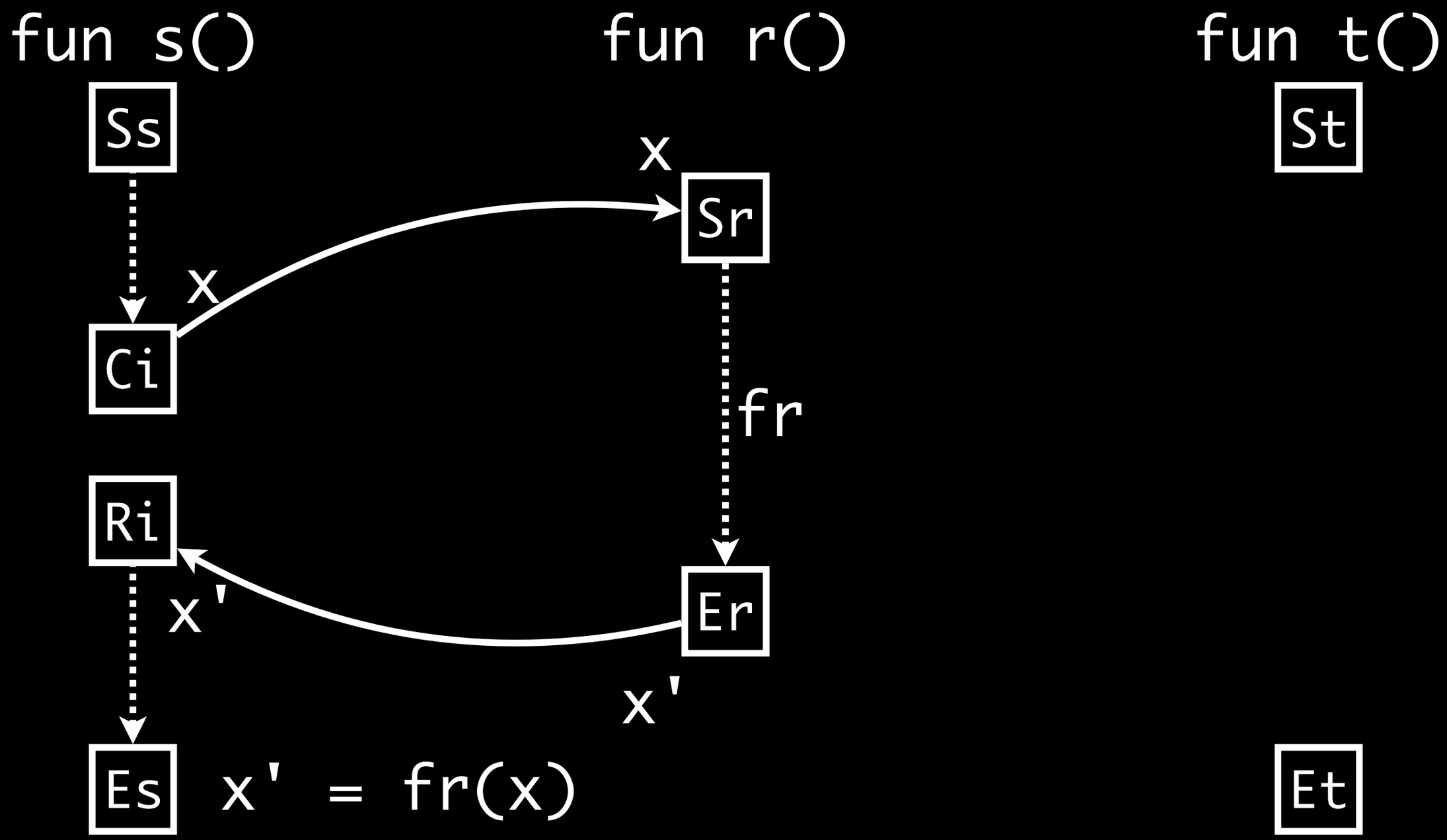
Inter-Procedural Data-Flow Analysis



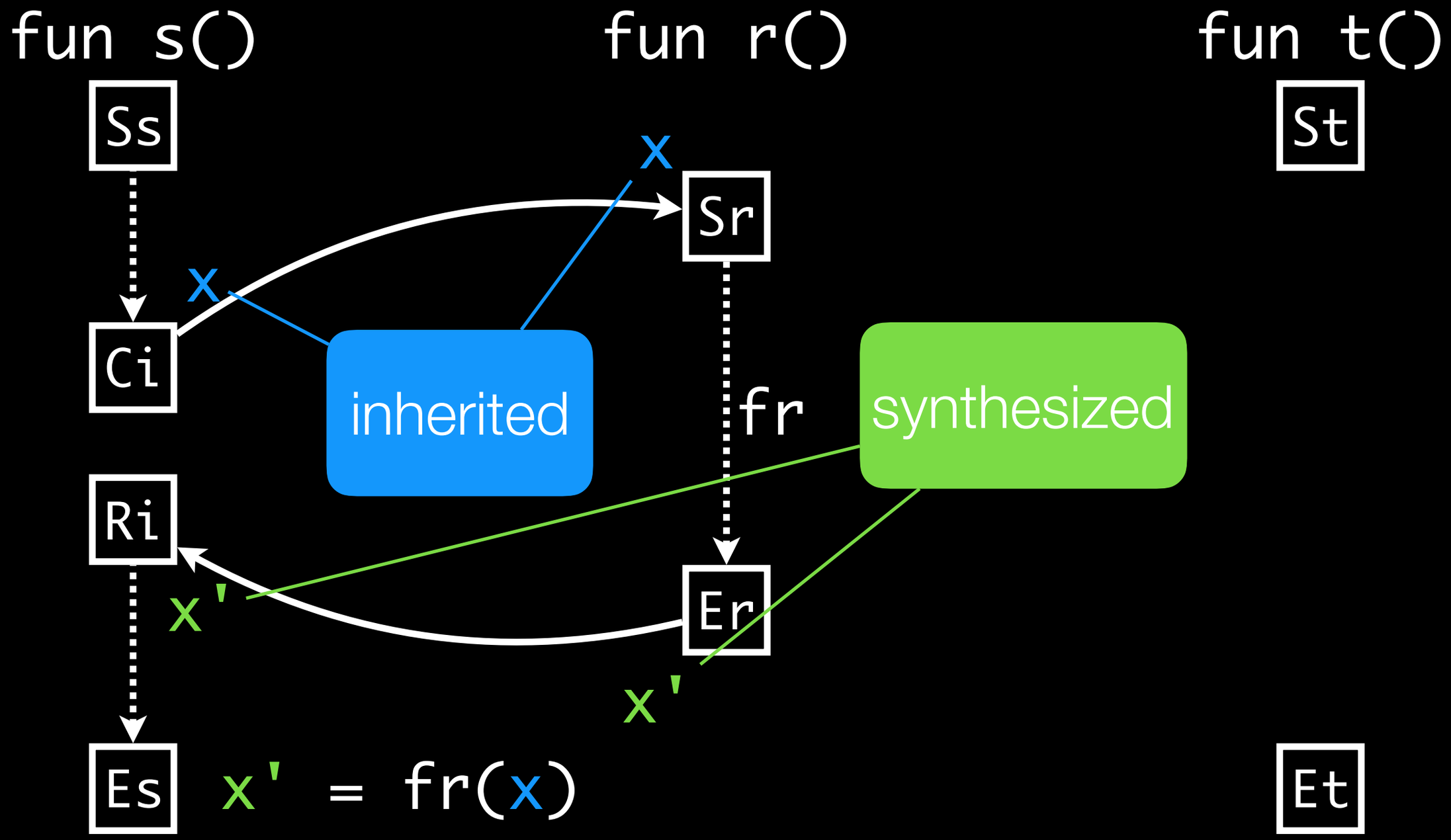
Inter-Procedural Data-Flow Analysis



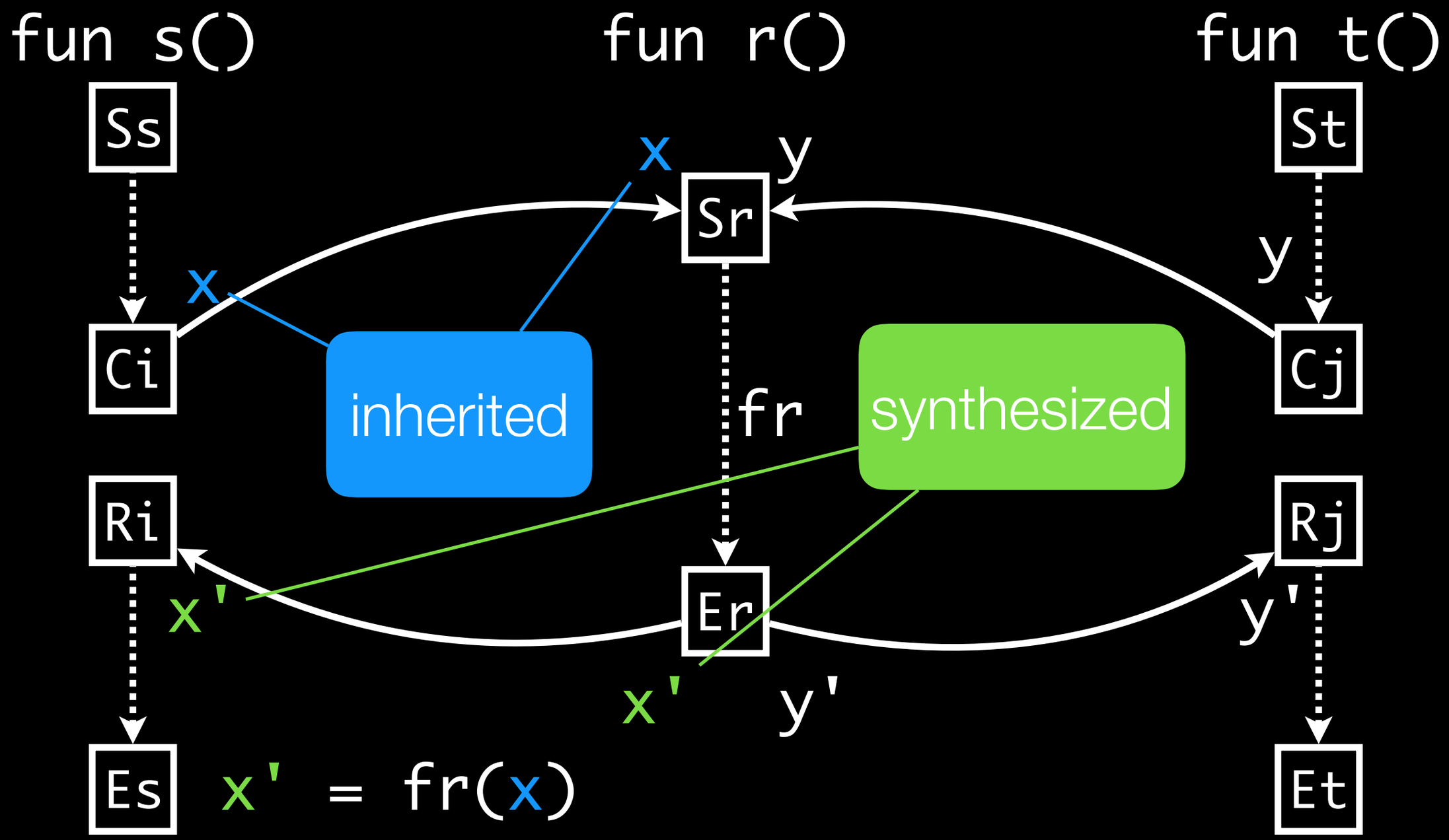
Inter-Procedural Data-Flow Analysis



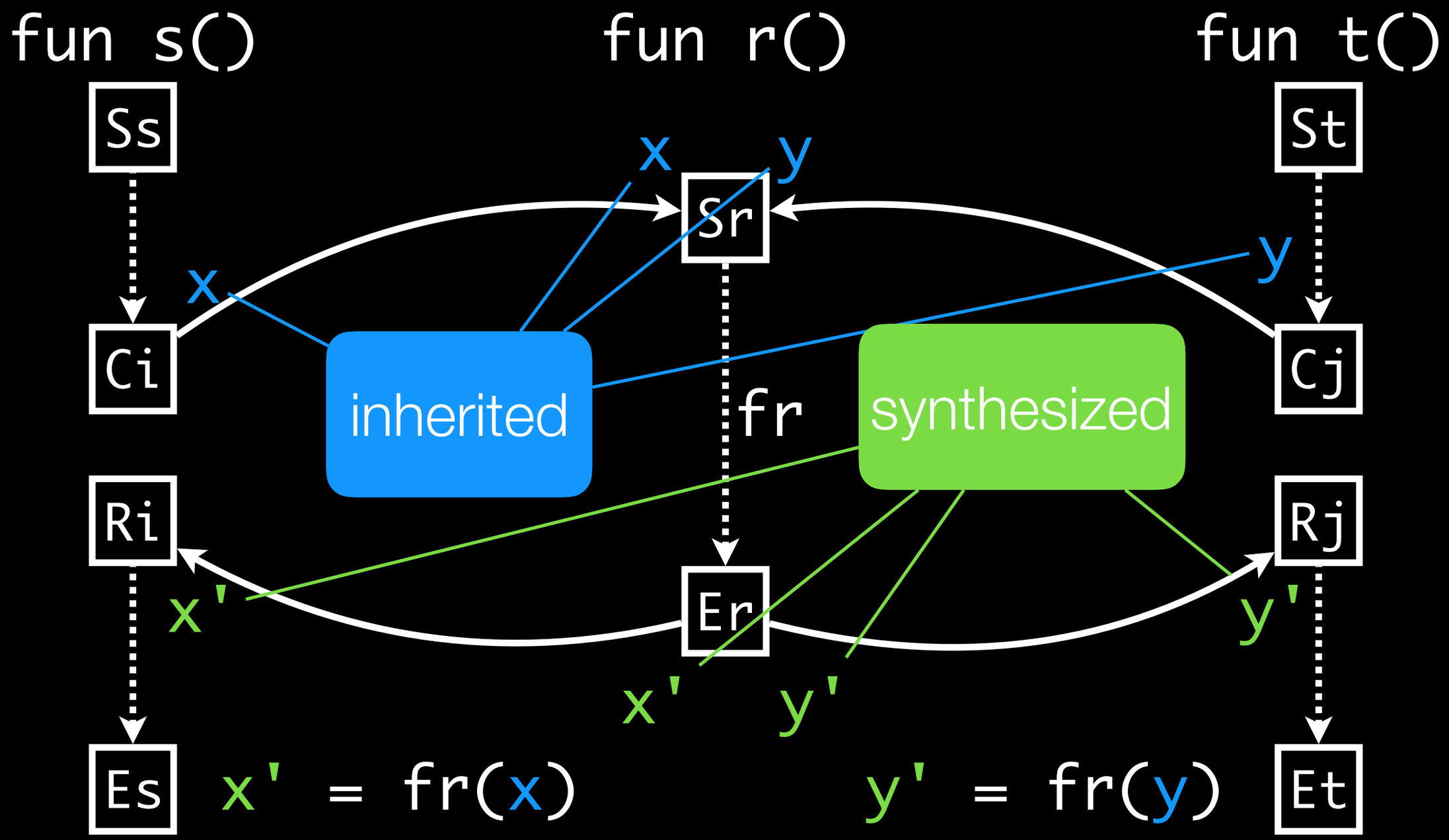
Inter-Procedural Data-Flow Analysis



Inter-Procedural Data-Flow Analysis



Inter-Procedural Data-Flow Analysis

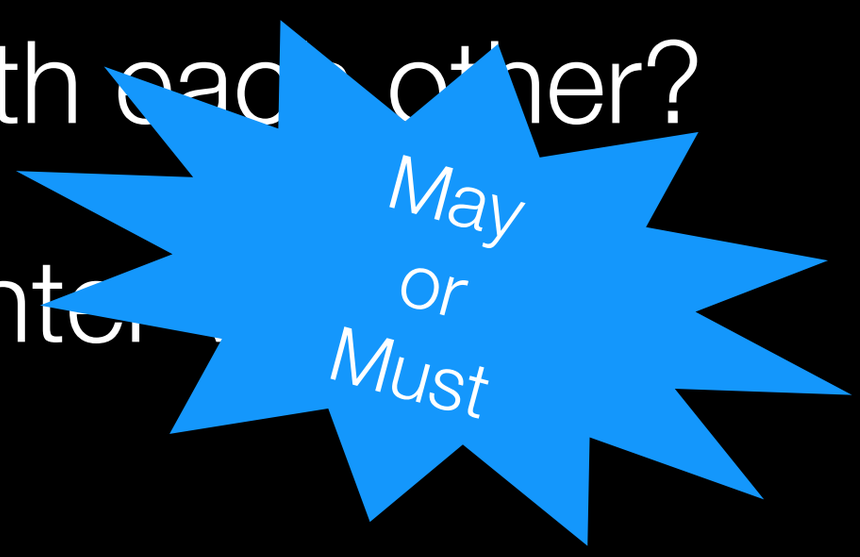


Inherited vs Synthesized

Analysis Information

Inherited Analysis Information

- Answering questions about formal parameters and global variables:
  - Which variables carry constant values?
  - Which variables aliased with each other?
  - Which locations can a pointer point to?



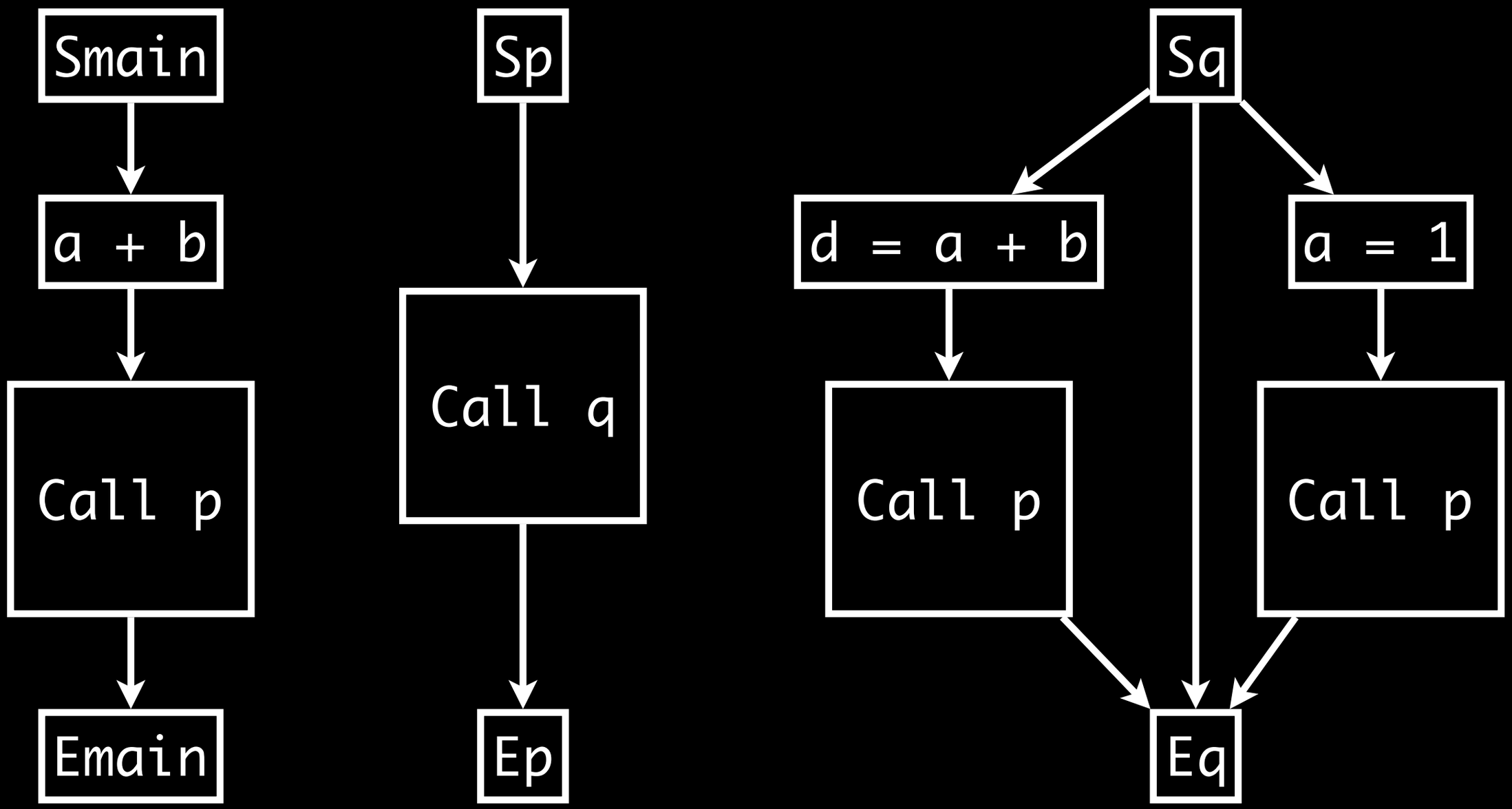
Synthesized Analysis Information

- Answering questions about side-effects of a procedure call:
  - Which local/global/formal variables are defined in a callee?
  - Which local/global/formal variables are used by a callee?

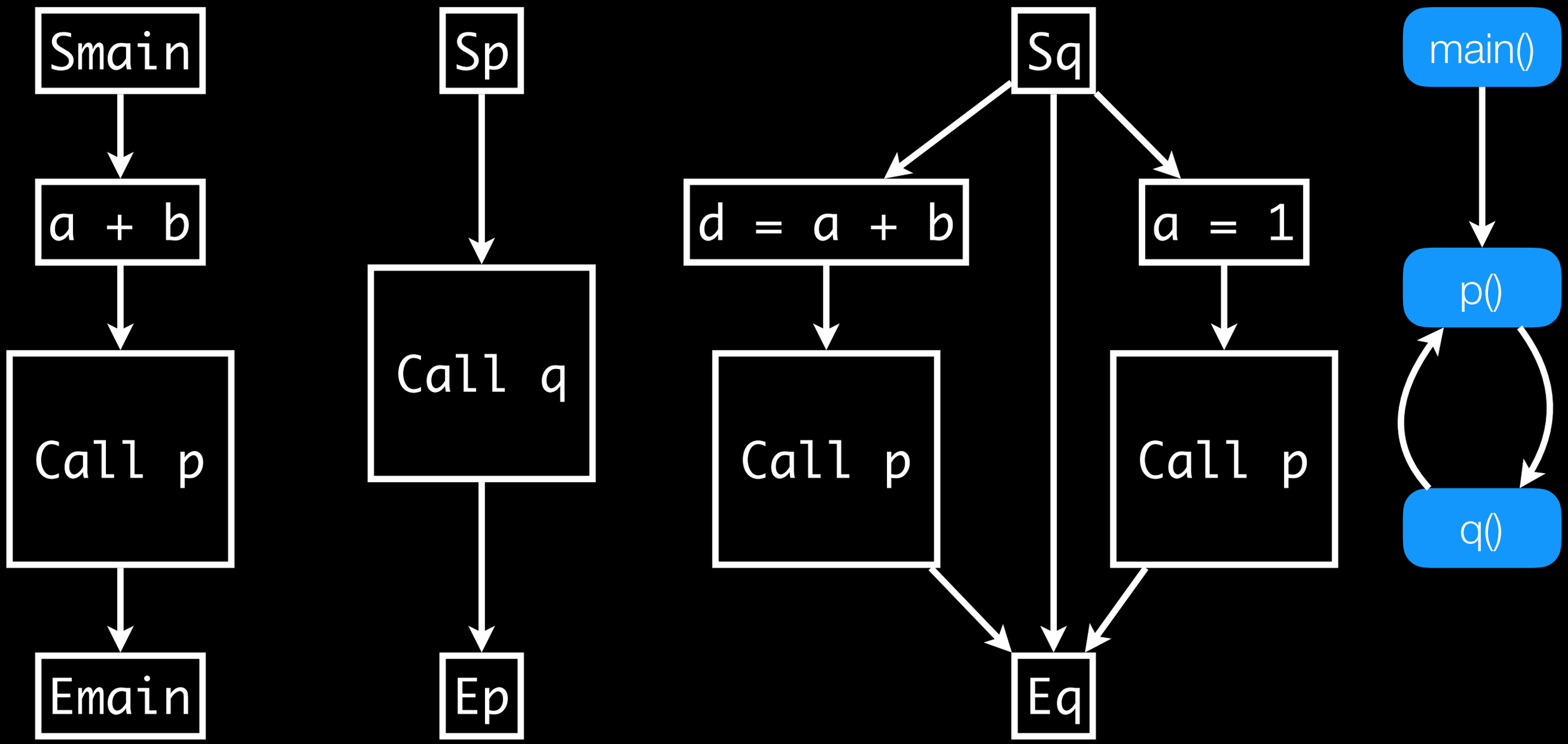
Inter-Procedural Control-Flow Graph (ICFG)

aka “program super-graph”

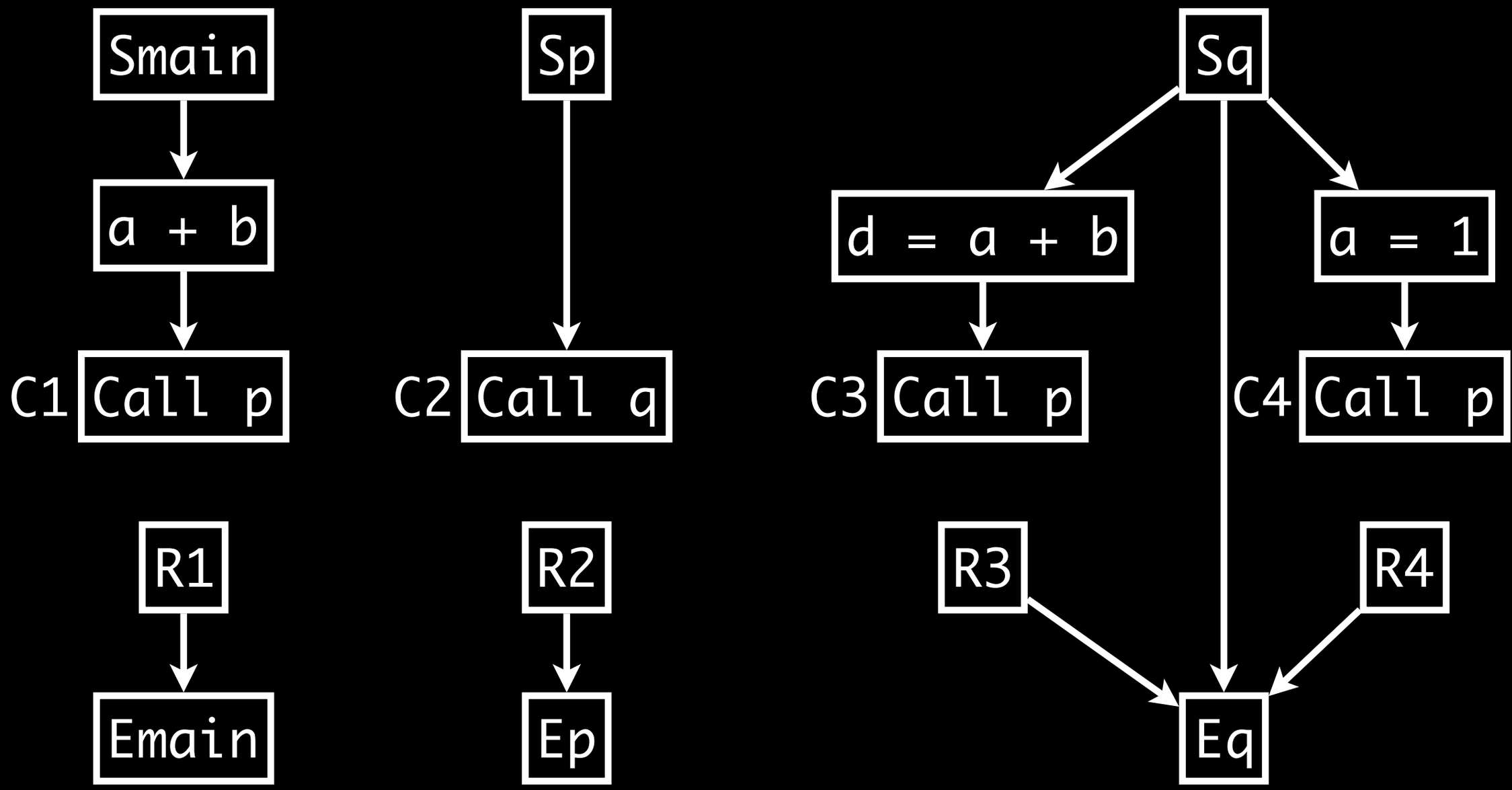
Procedure Space



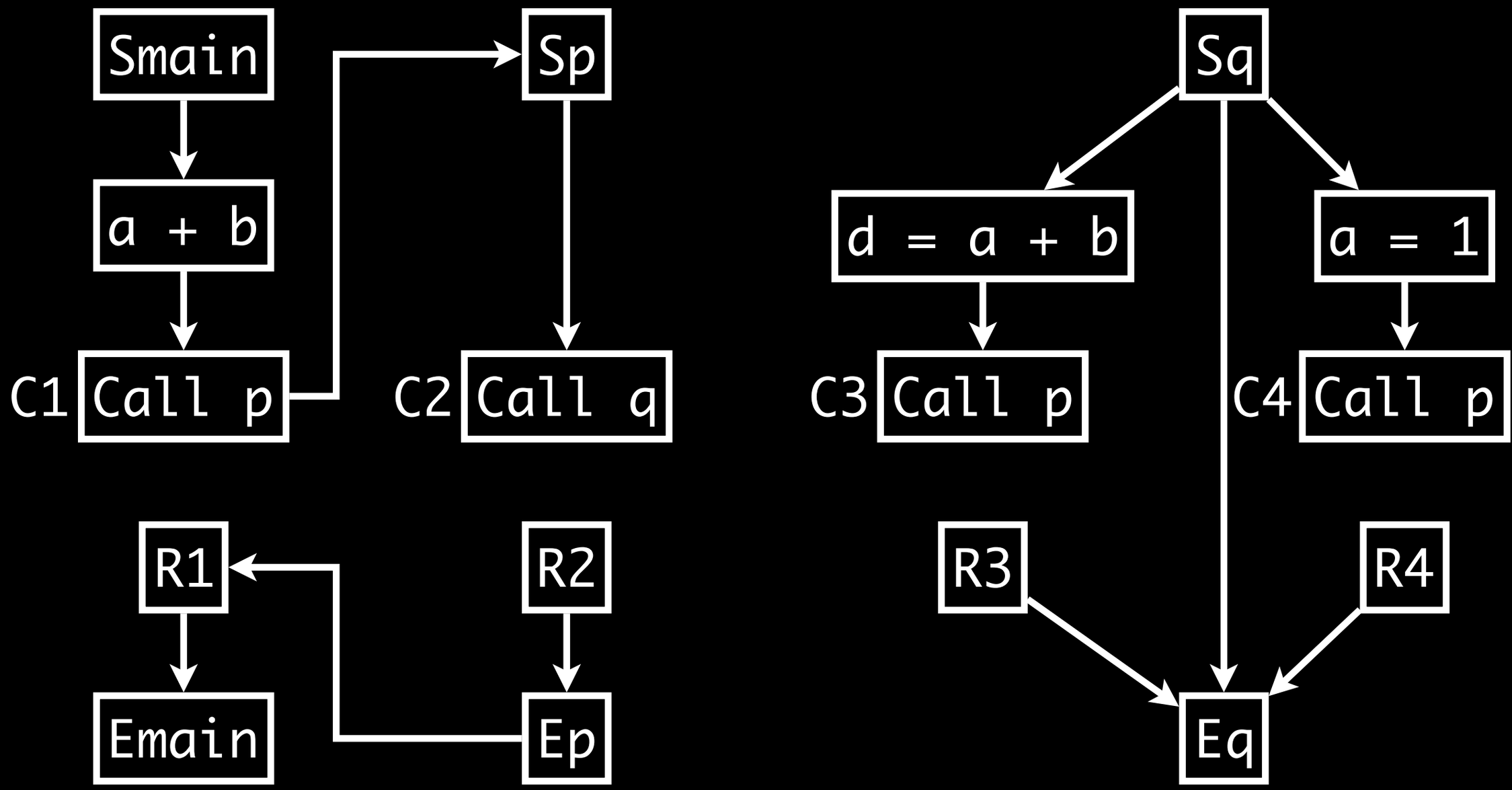
Call Graph


Smain
a + b
Call p
Emain
Sp
Call q
Ep
Sq
d = a + b
Call p
a = 1
Call p
Eq
main()
p()
q()

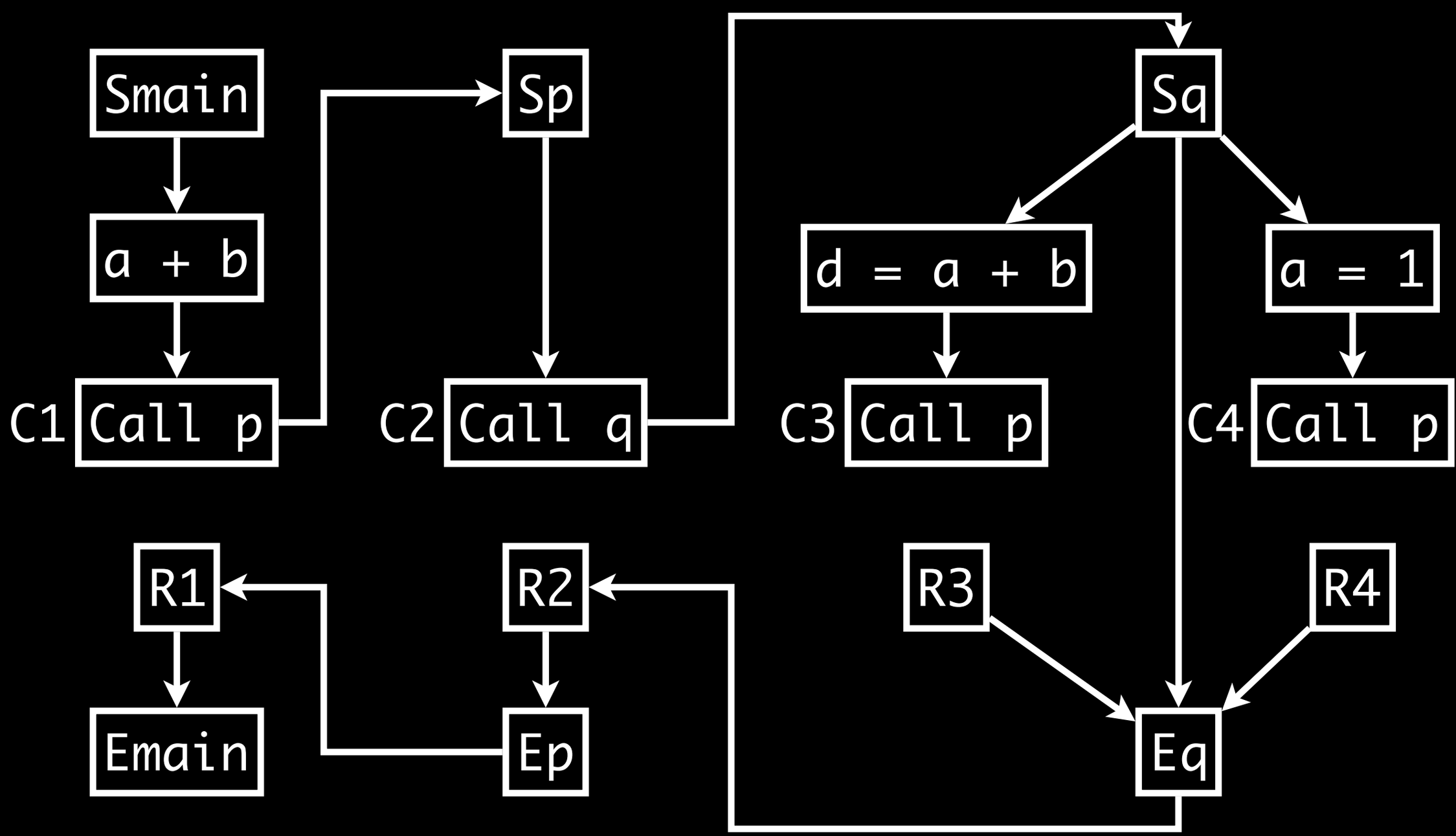
Introduce Return Sites



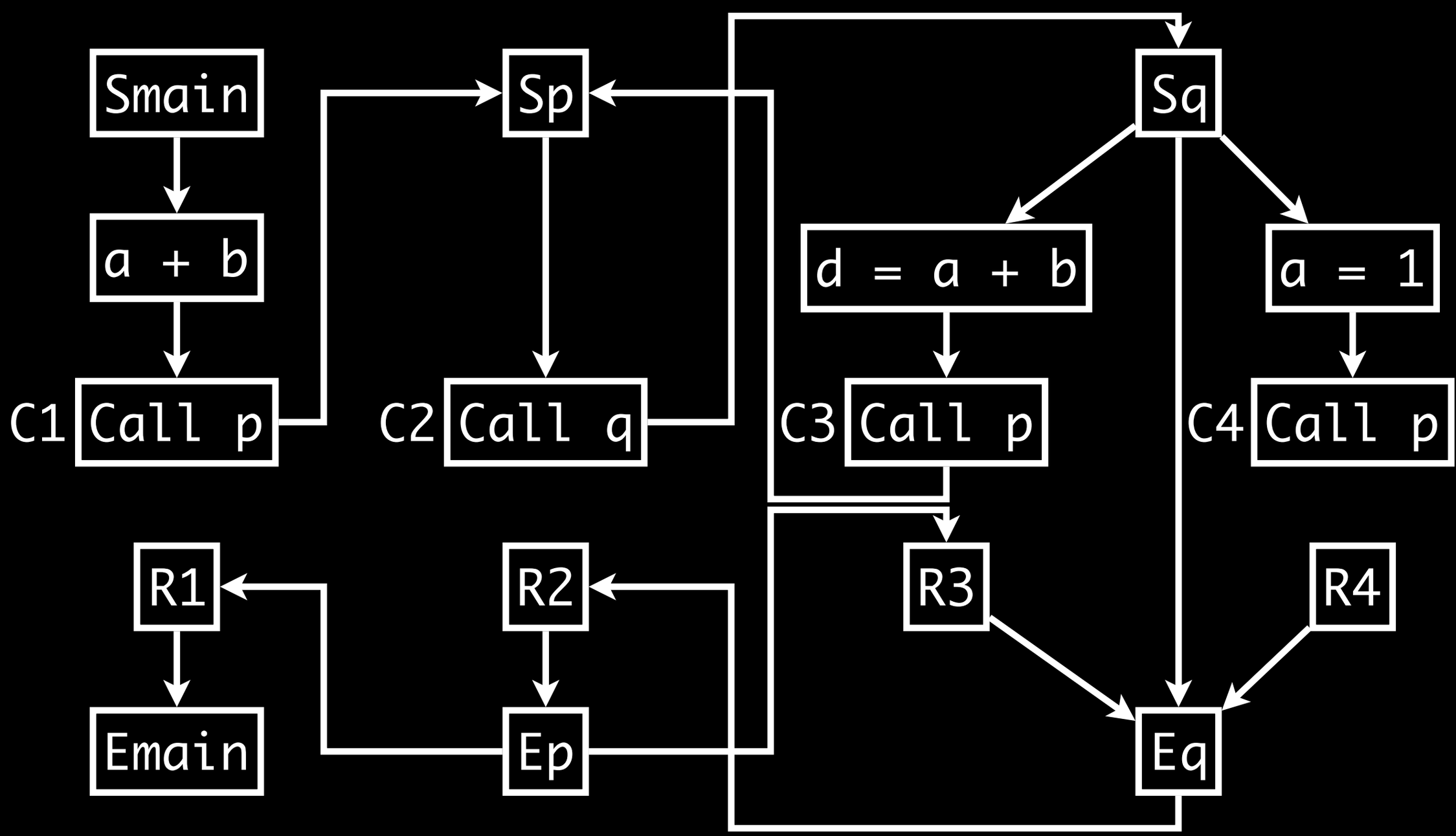
Caller-Callee Relationships



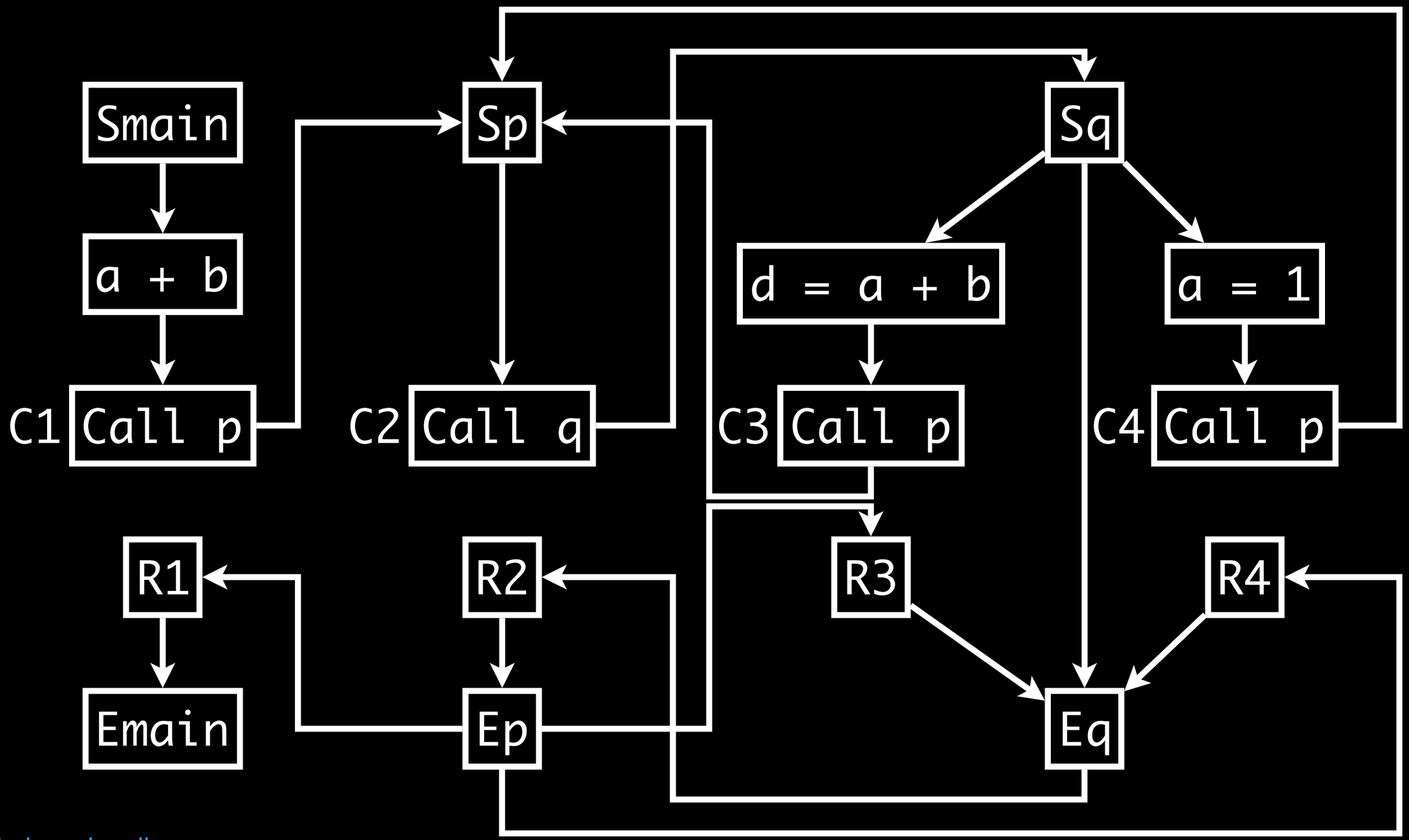
Caller-Callee Relationships



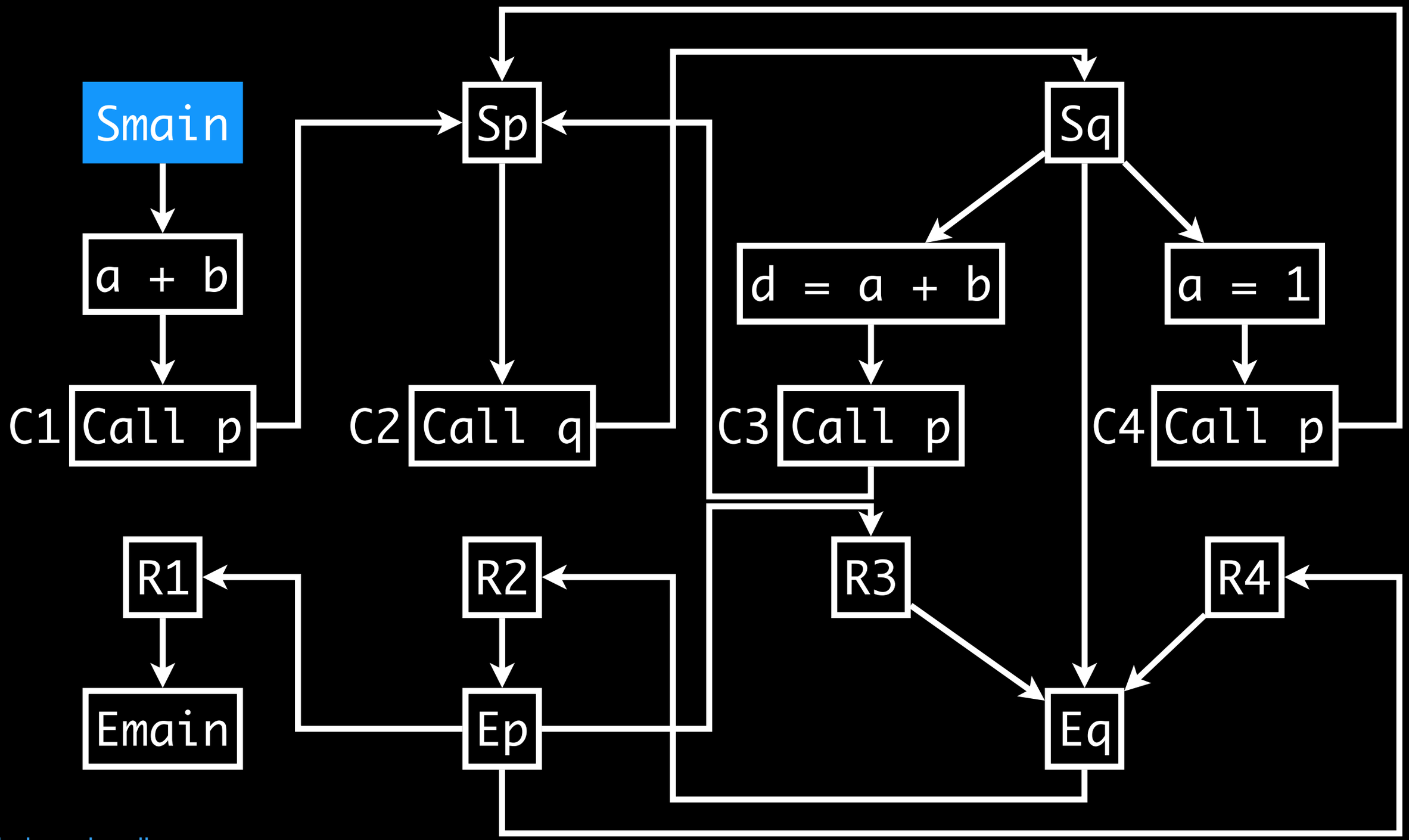
Caller-Callee Relationships



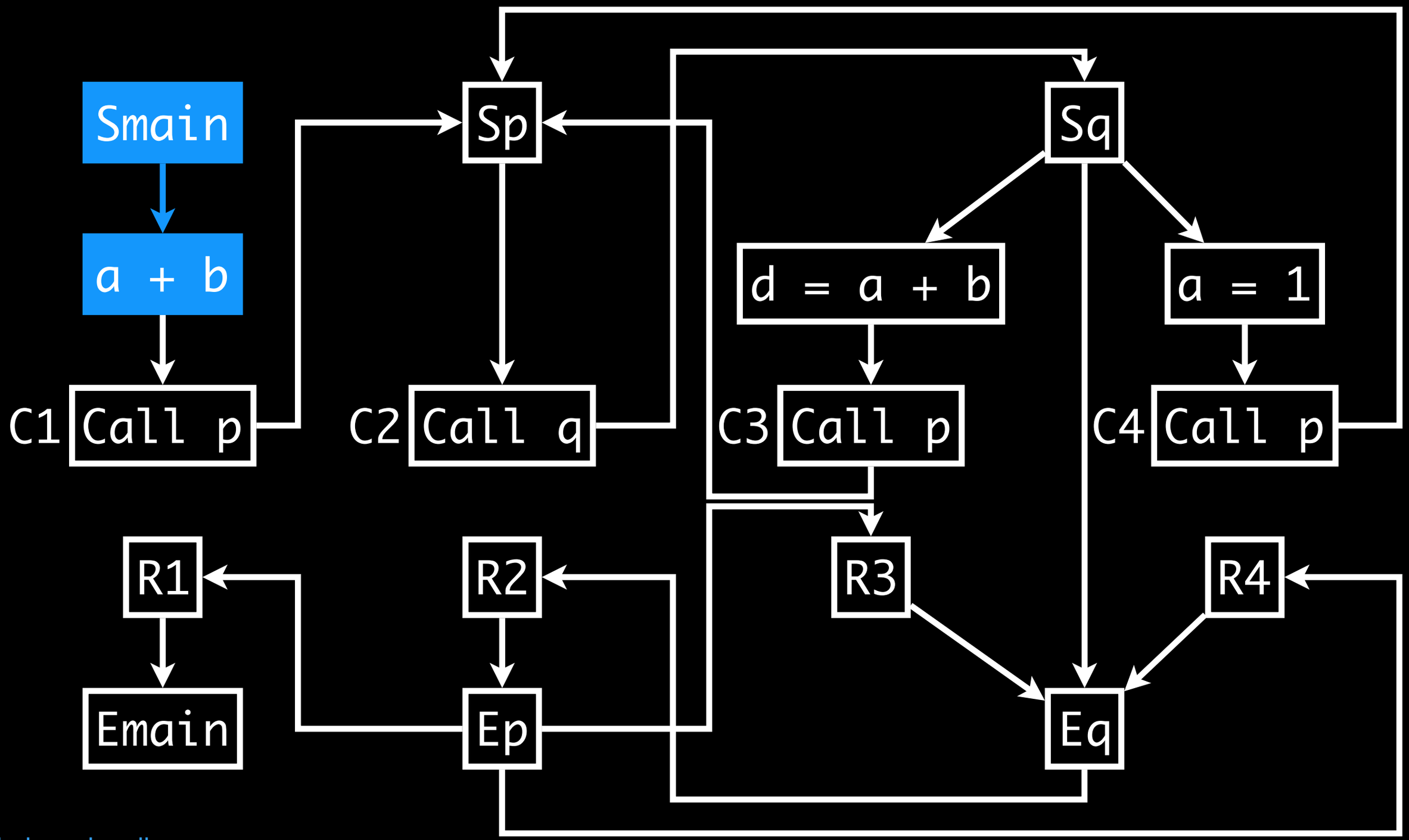
Caller-Callee Relationships



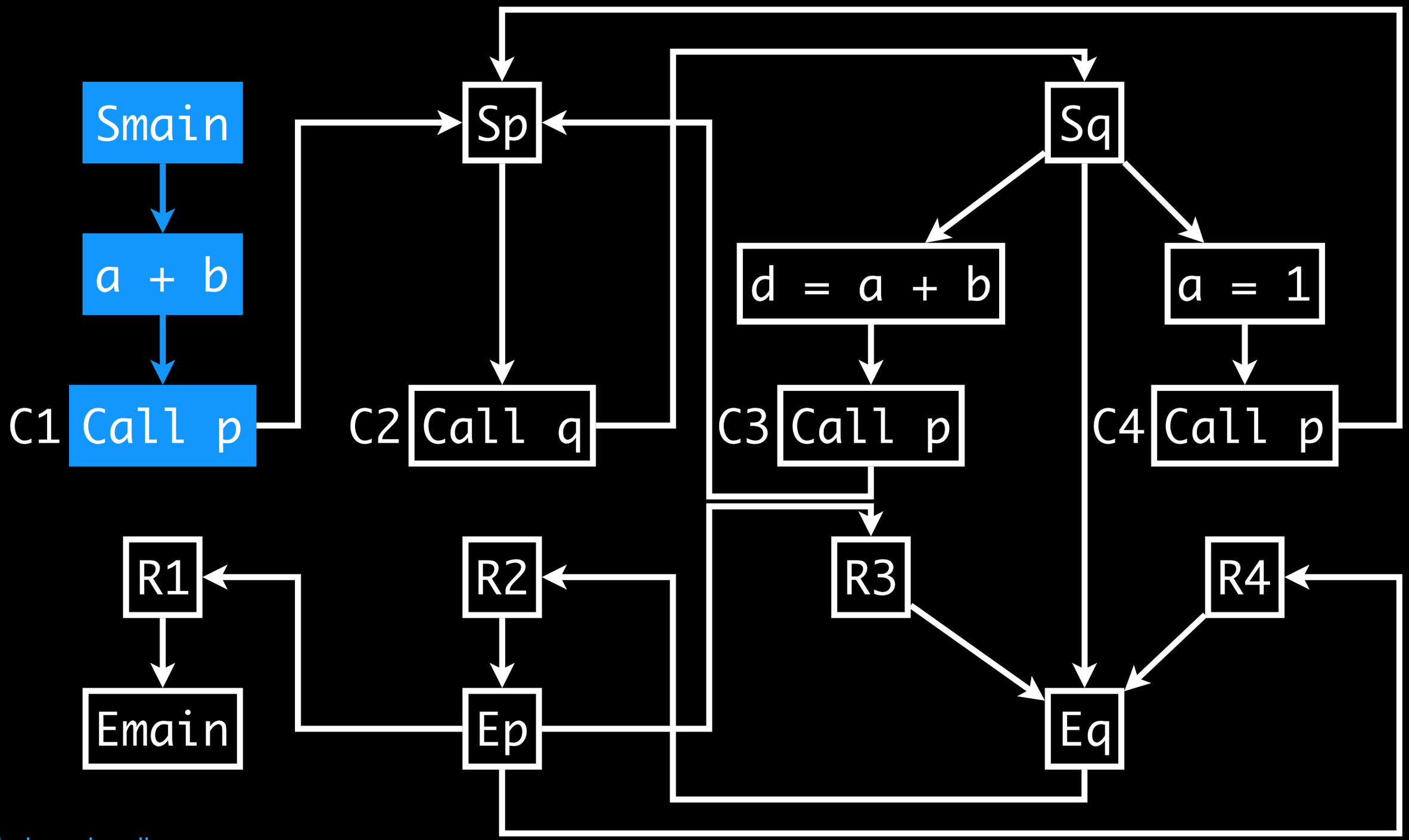
Valid/Realizable Path



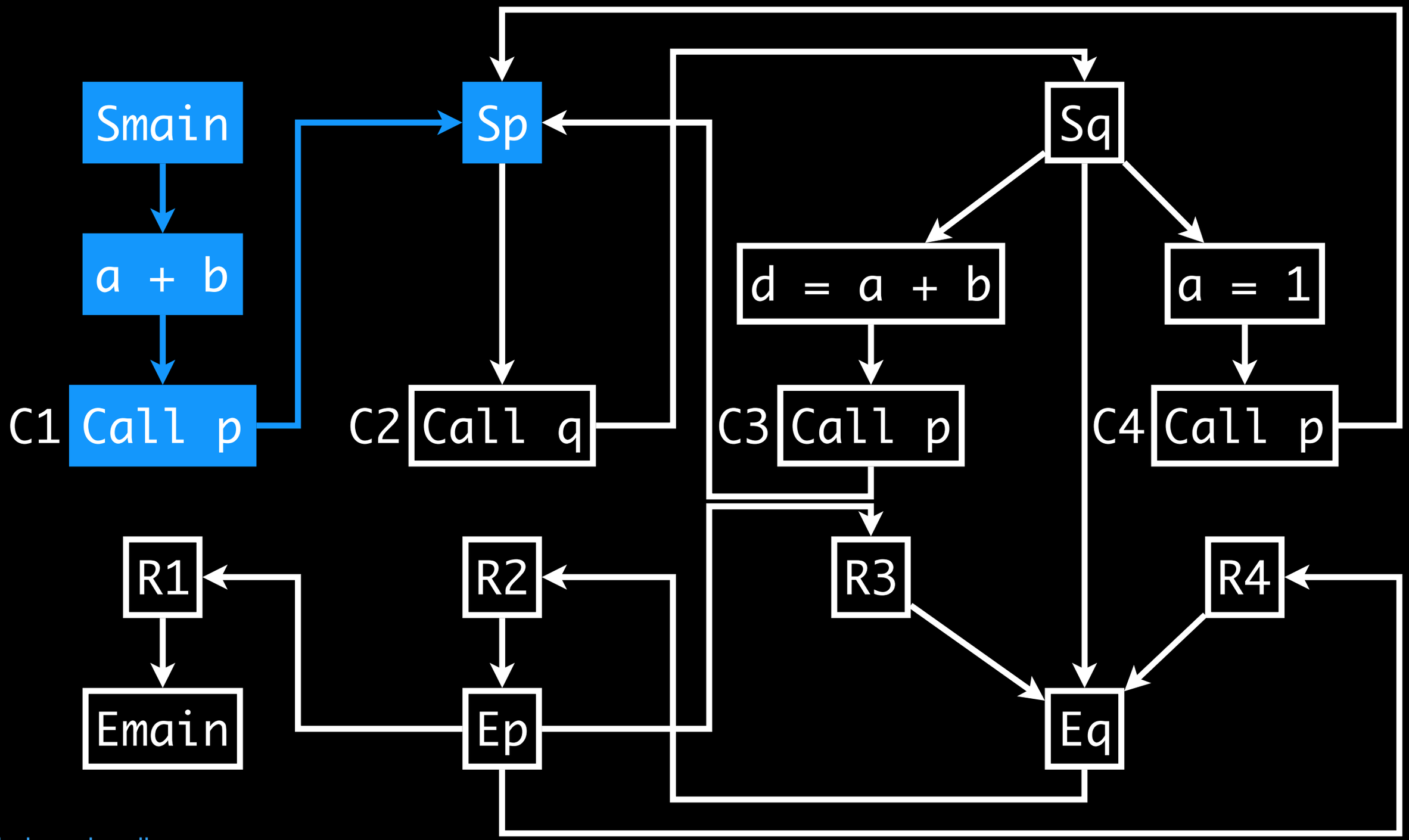
Valid/Realizable Path



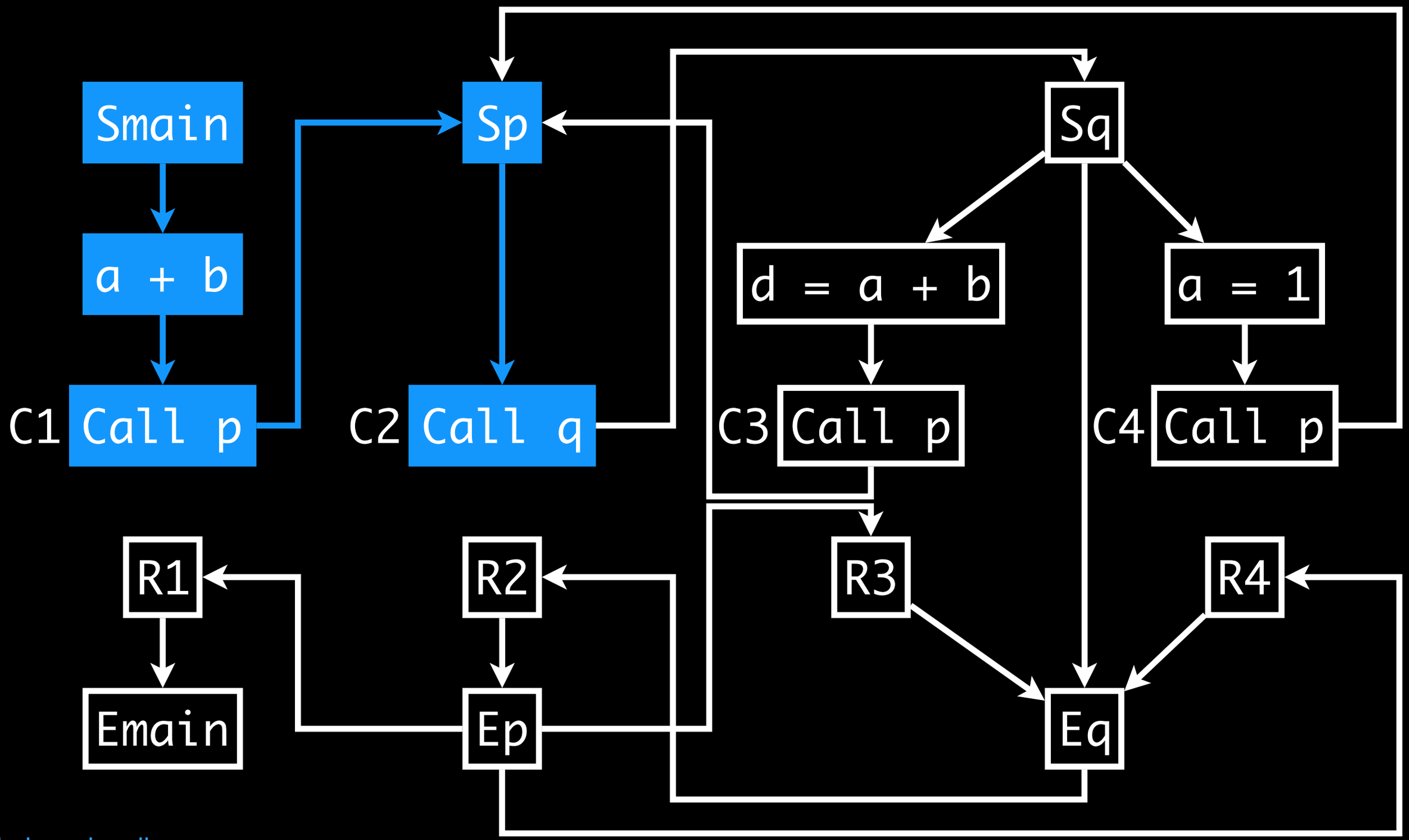
Valid/Realizable Path



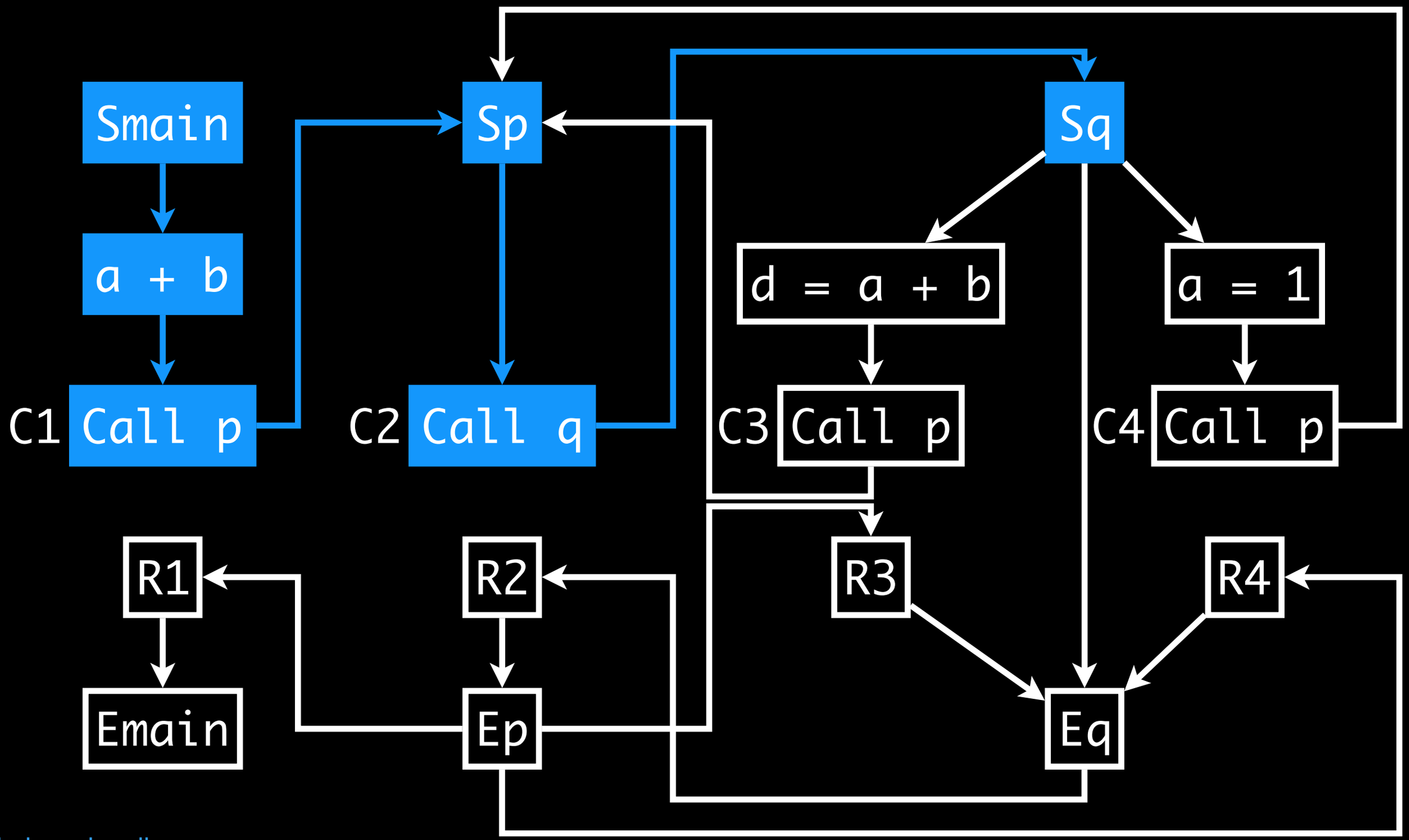
Valid/Realizable Path



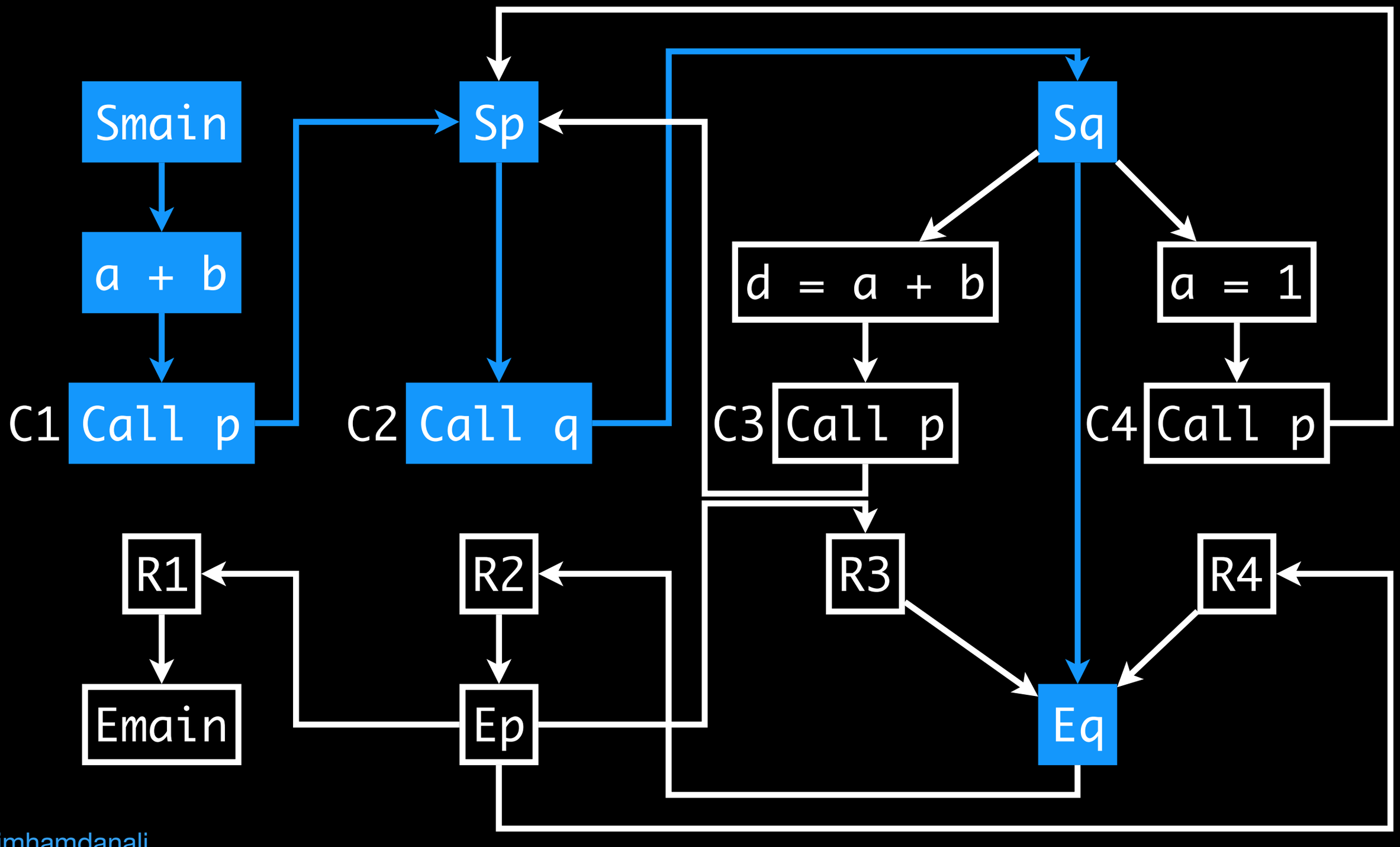
Valid/Realizable Path



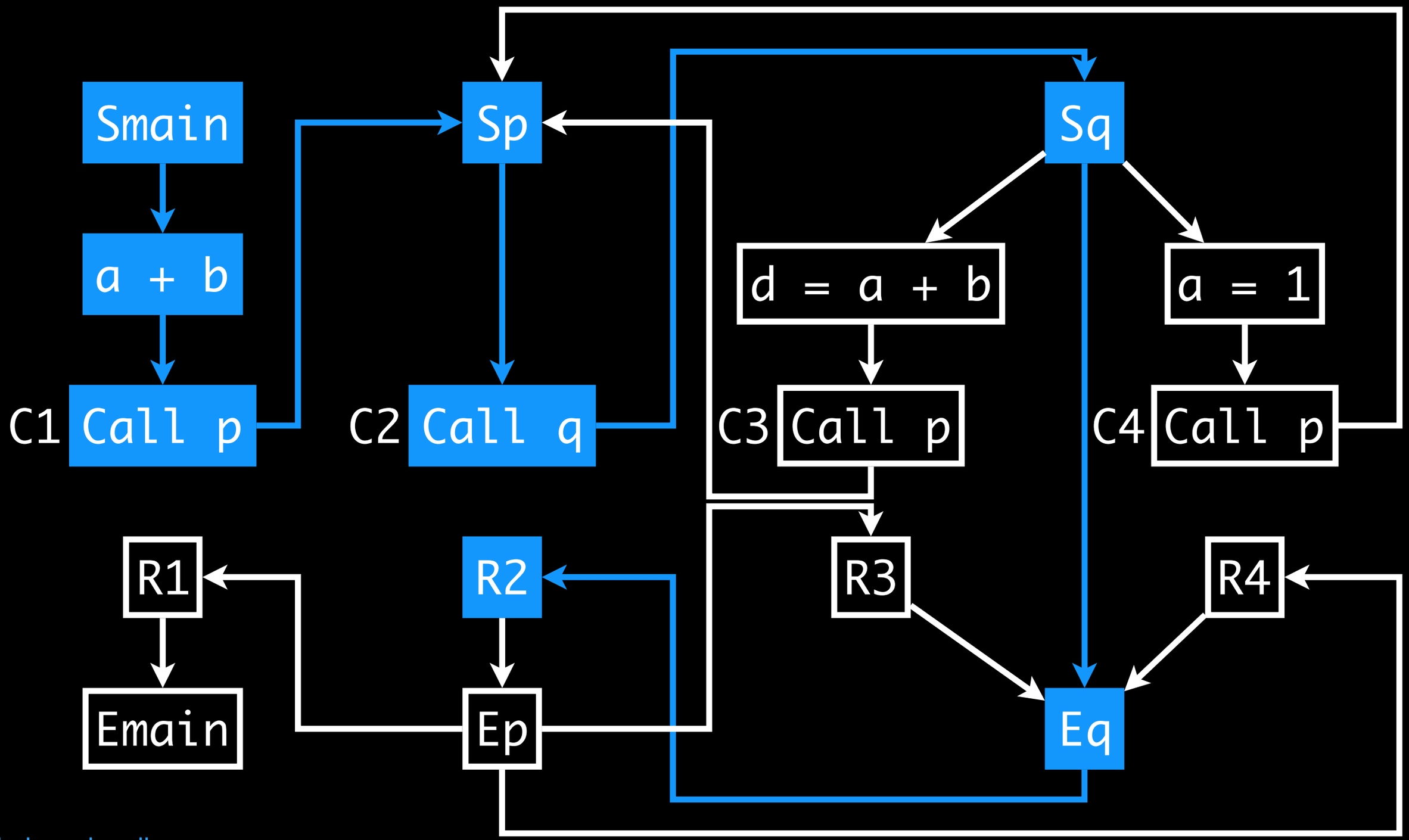
Valid/Realizable Path



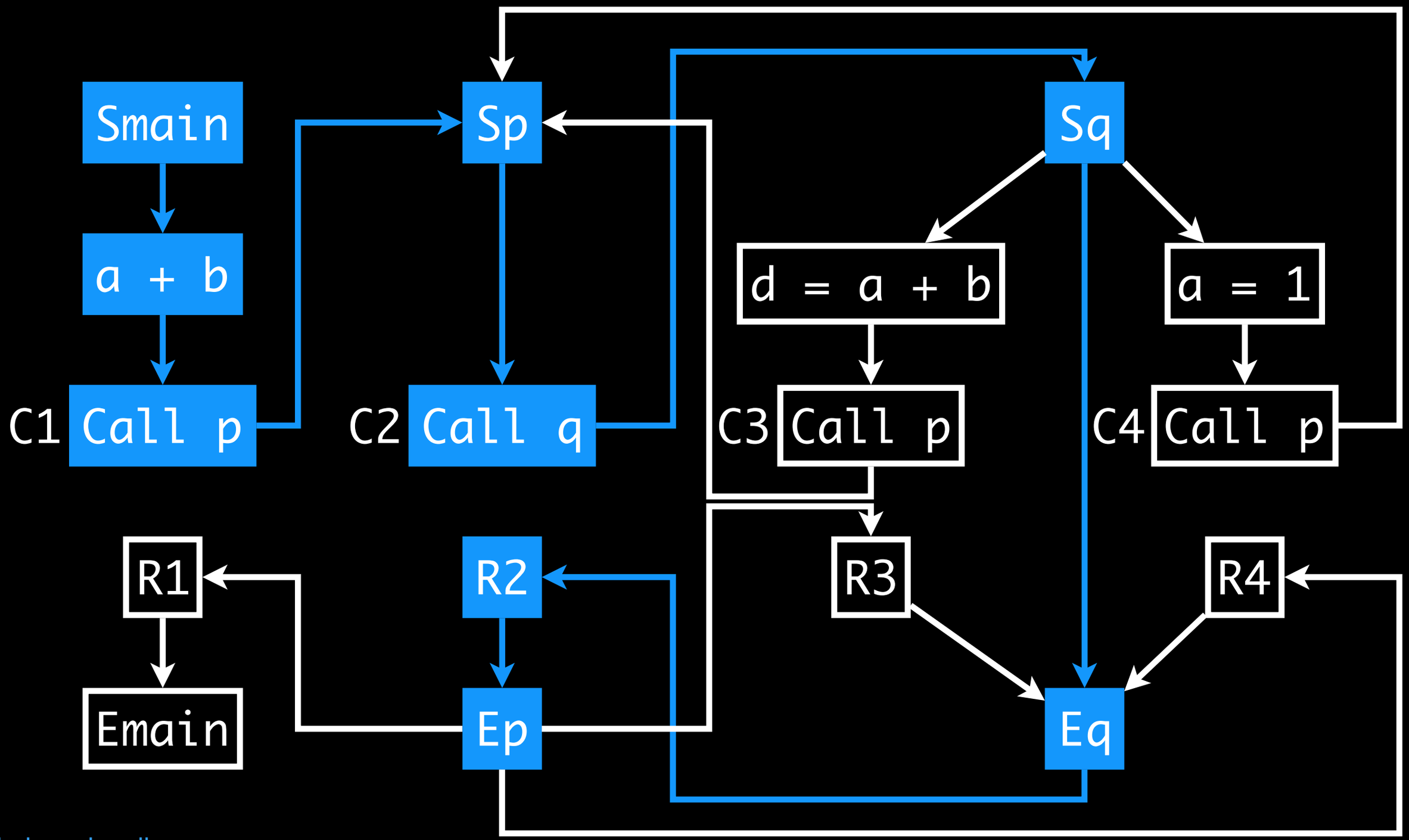
Valid/Realizable Path



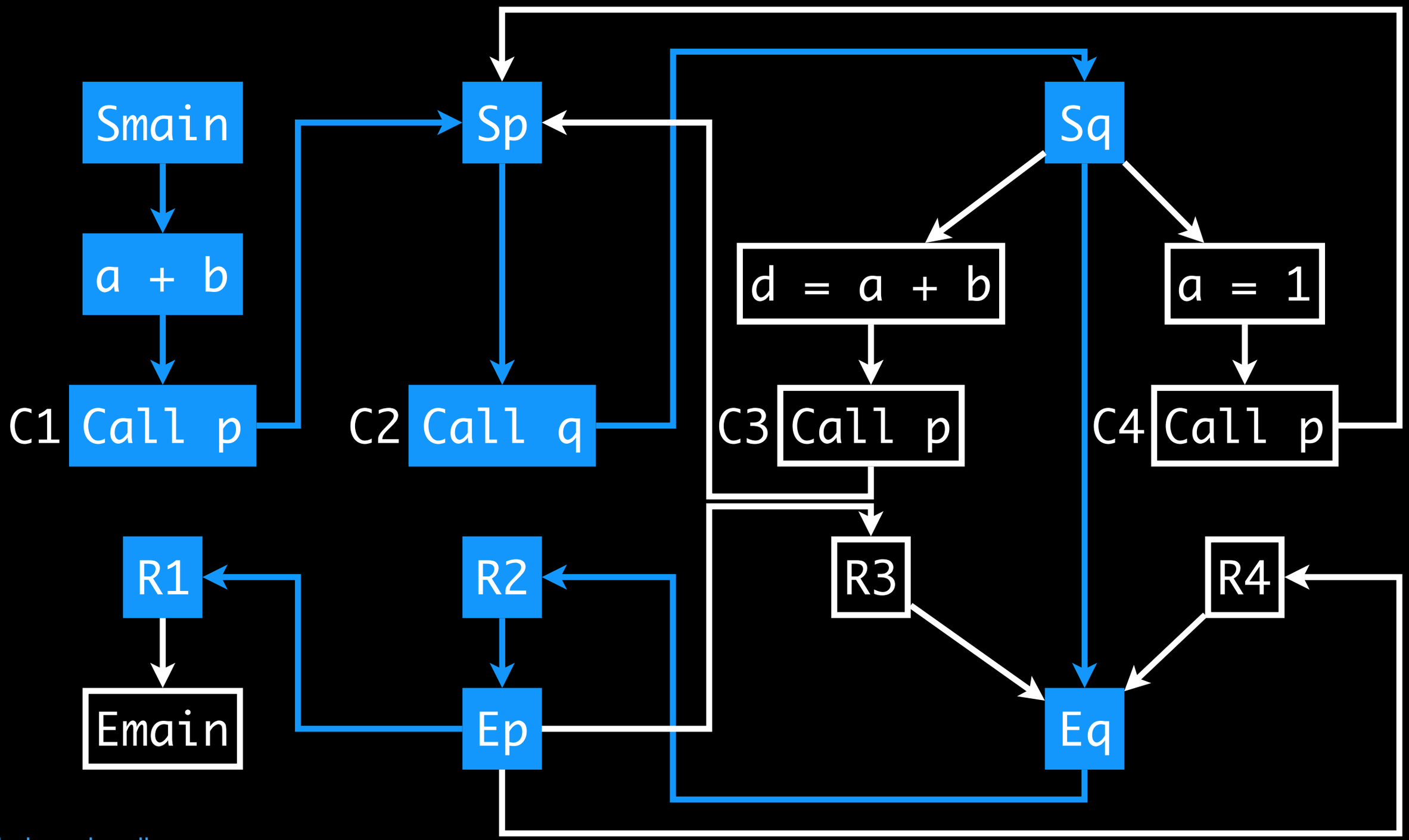
Valid/Realizable Path



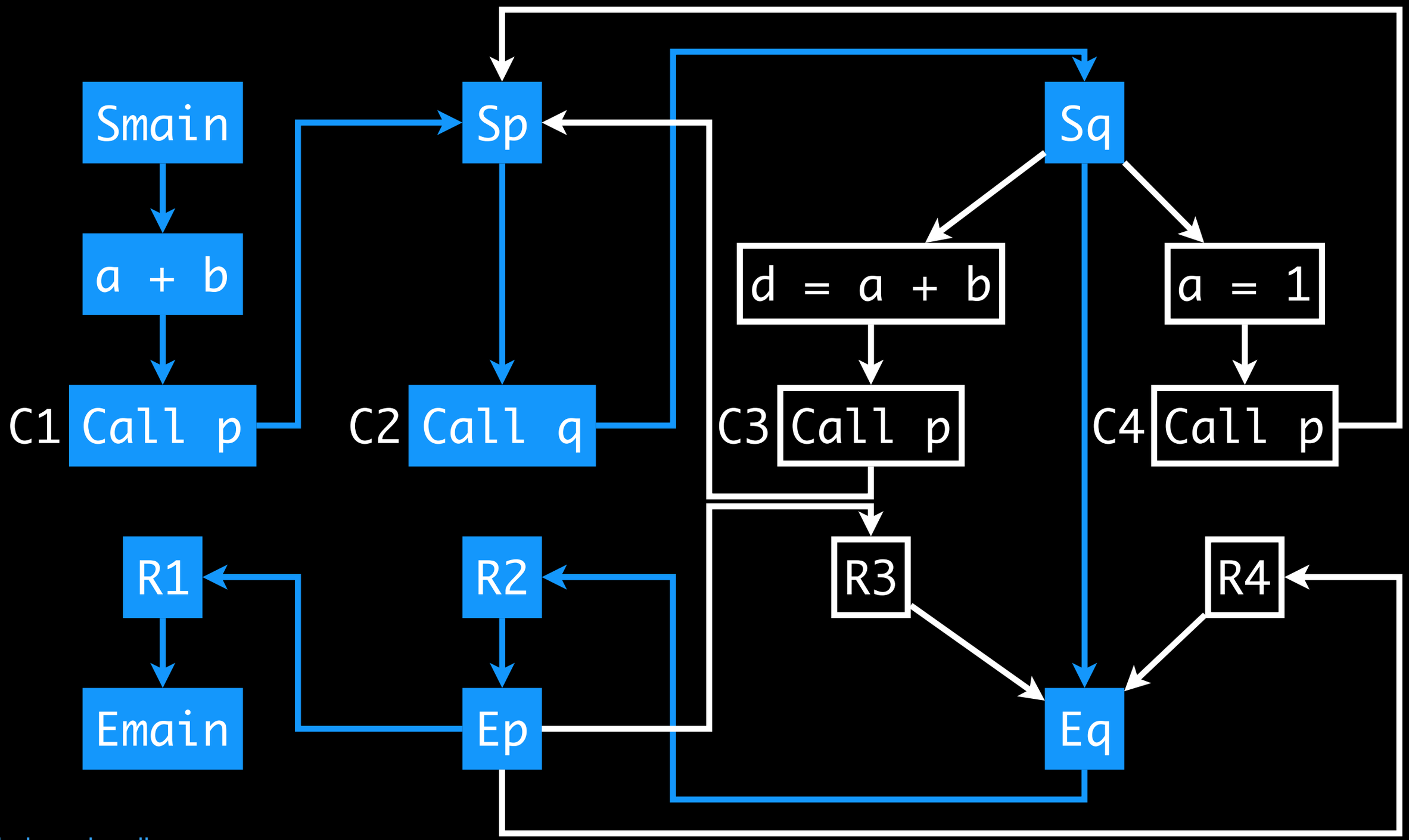
Valid/Realizable Path



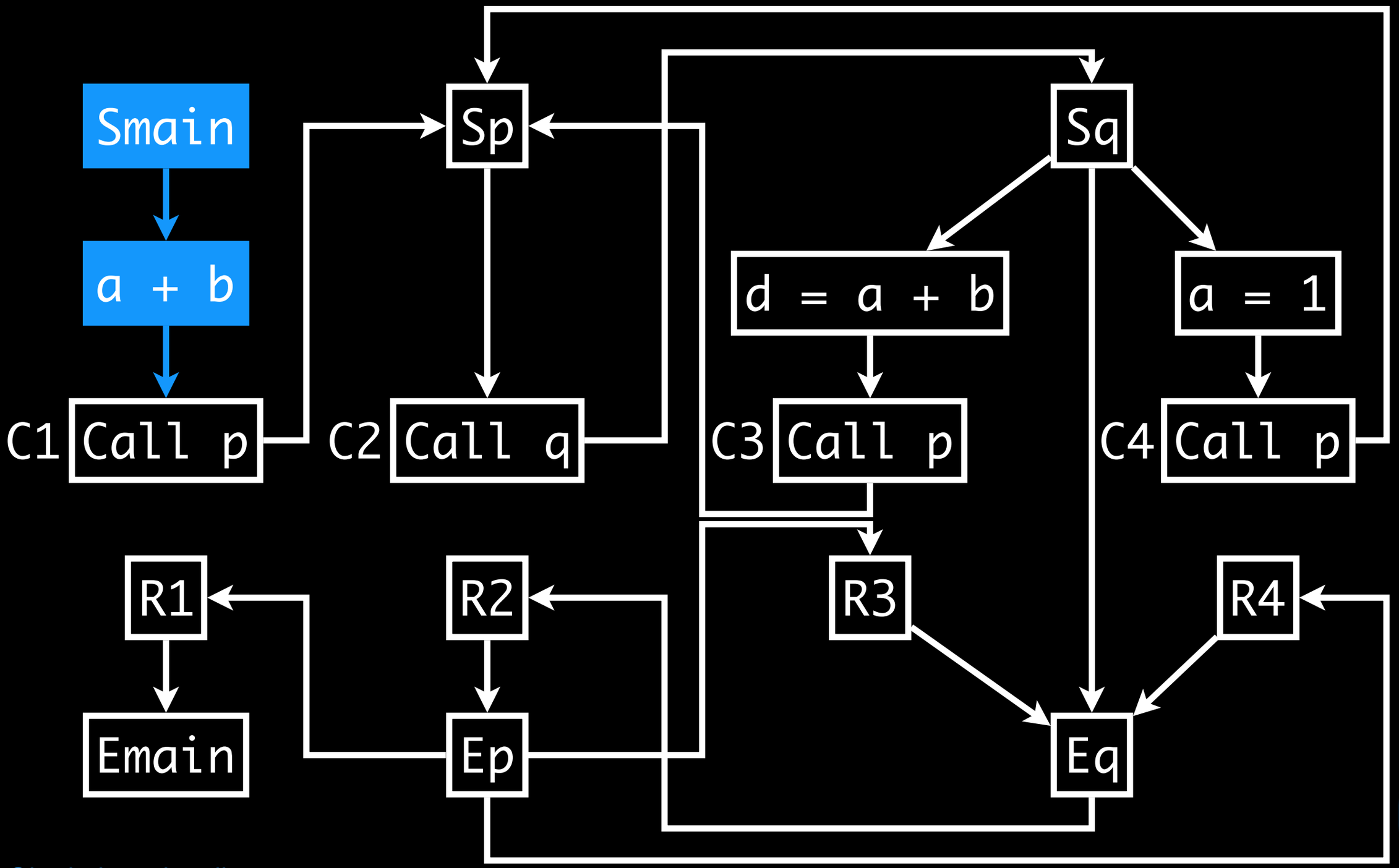
Valid/Realizable Path



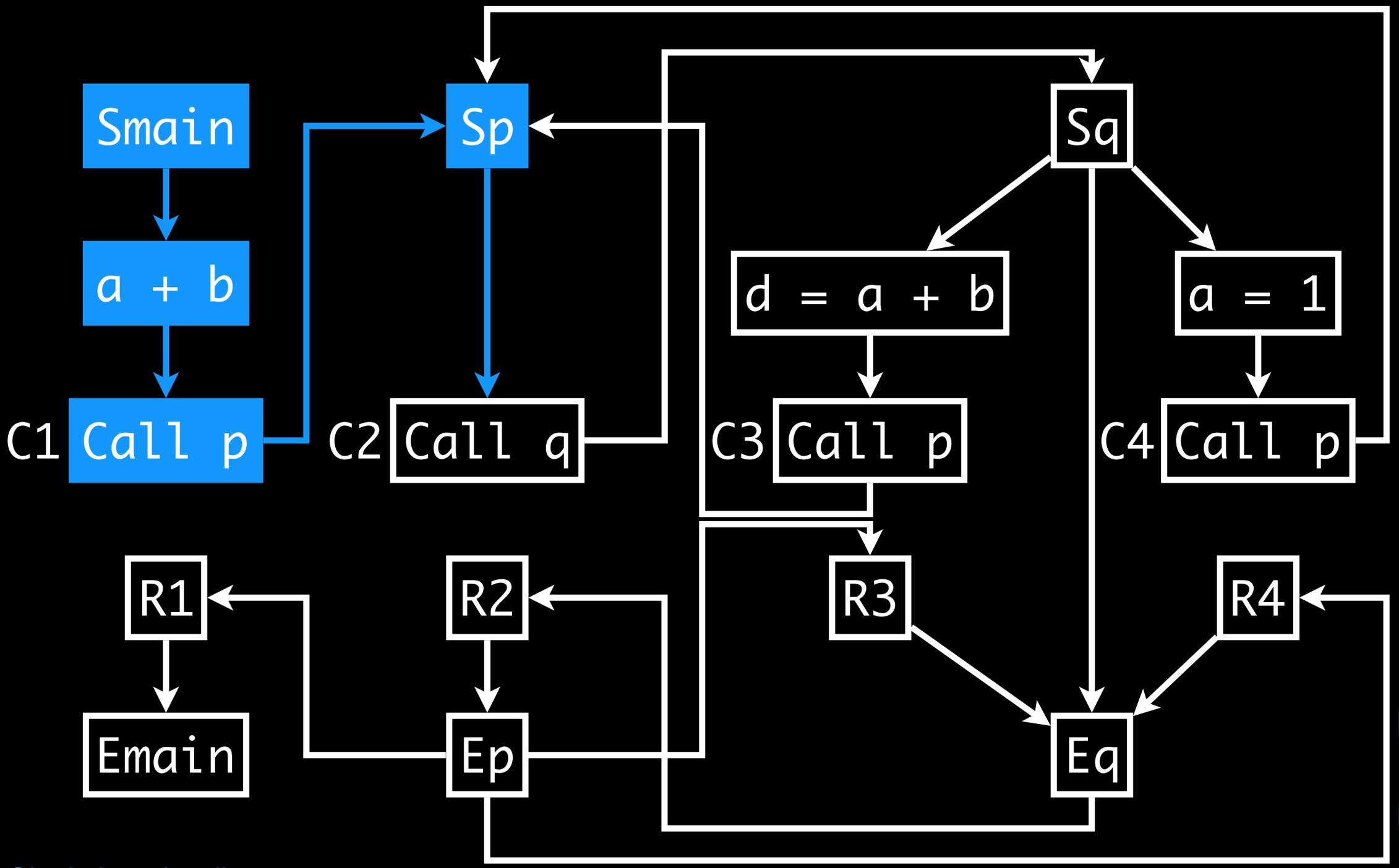
Valid/Realizable Path



Valid/Realizable Path



Valid/Realizable Path

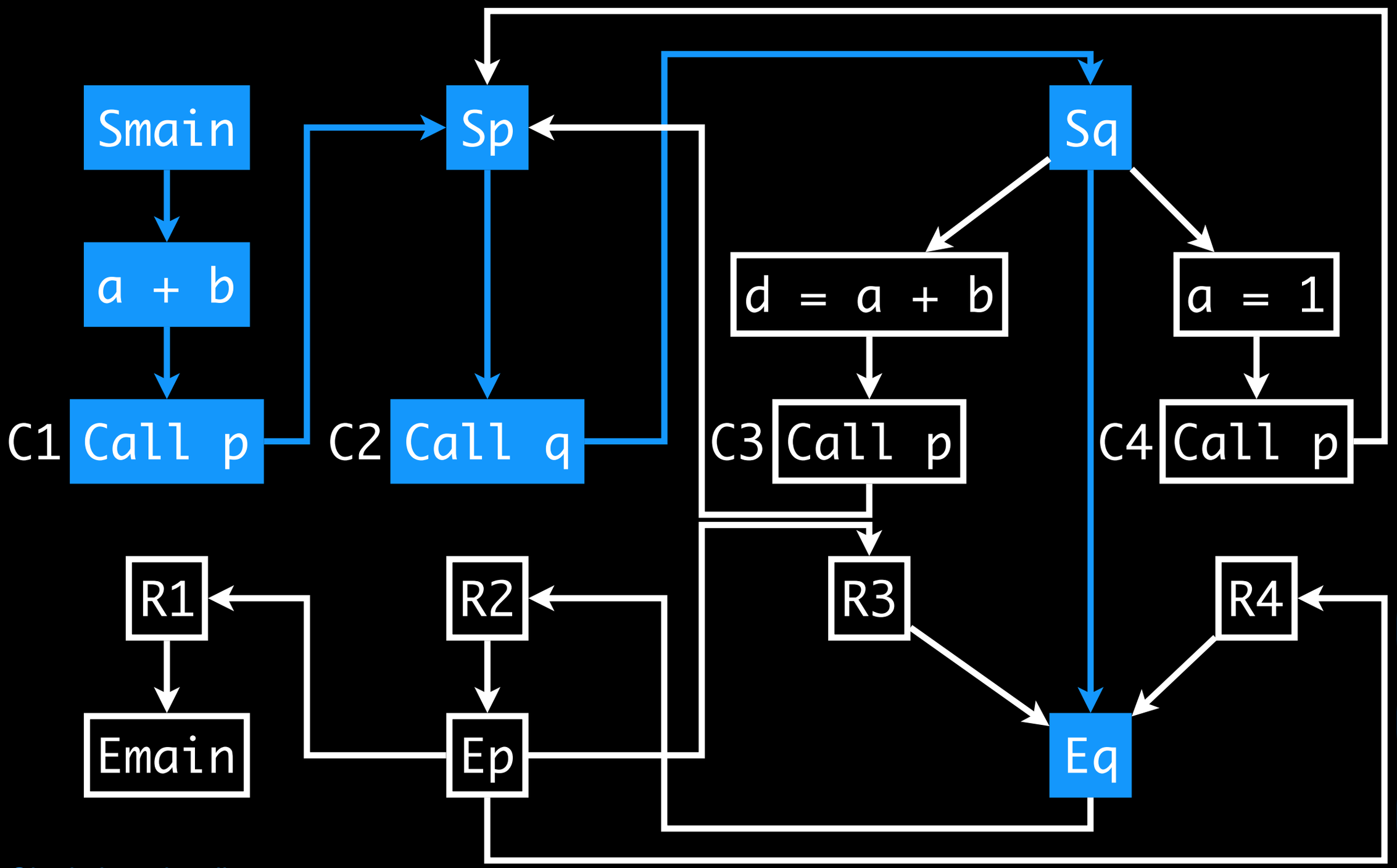


Call Stack

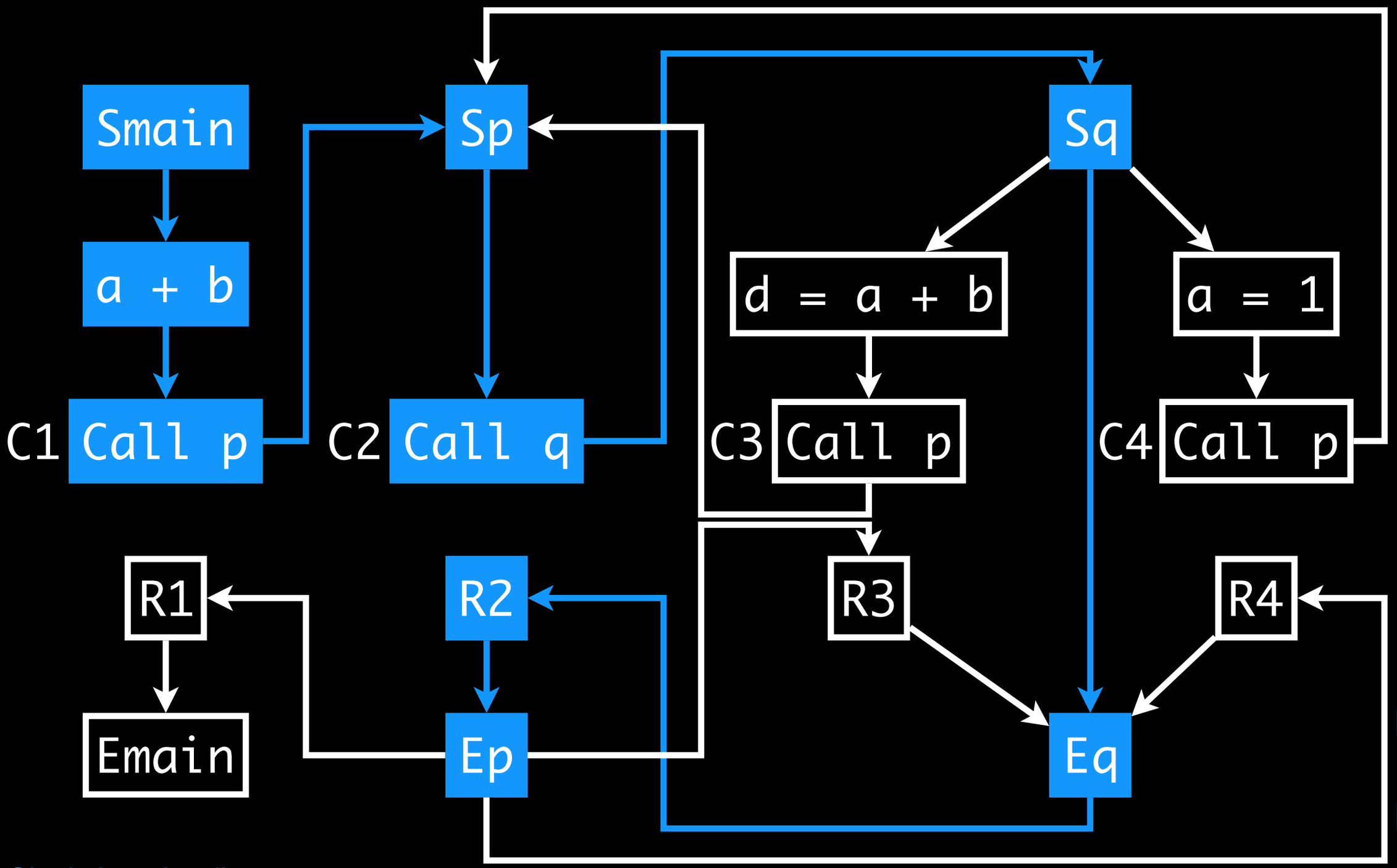
p : C1

main()

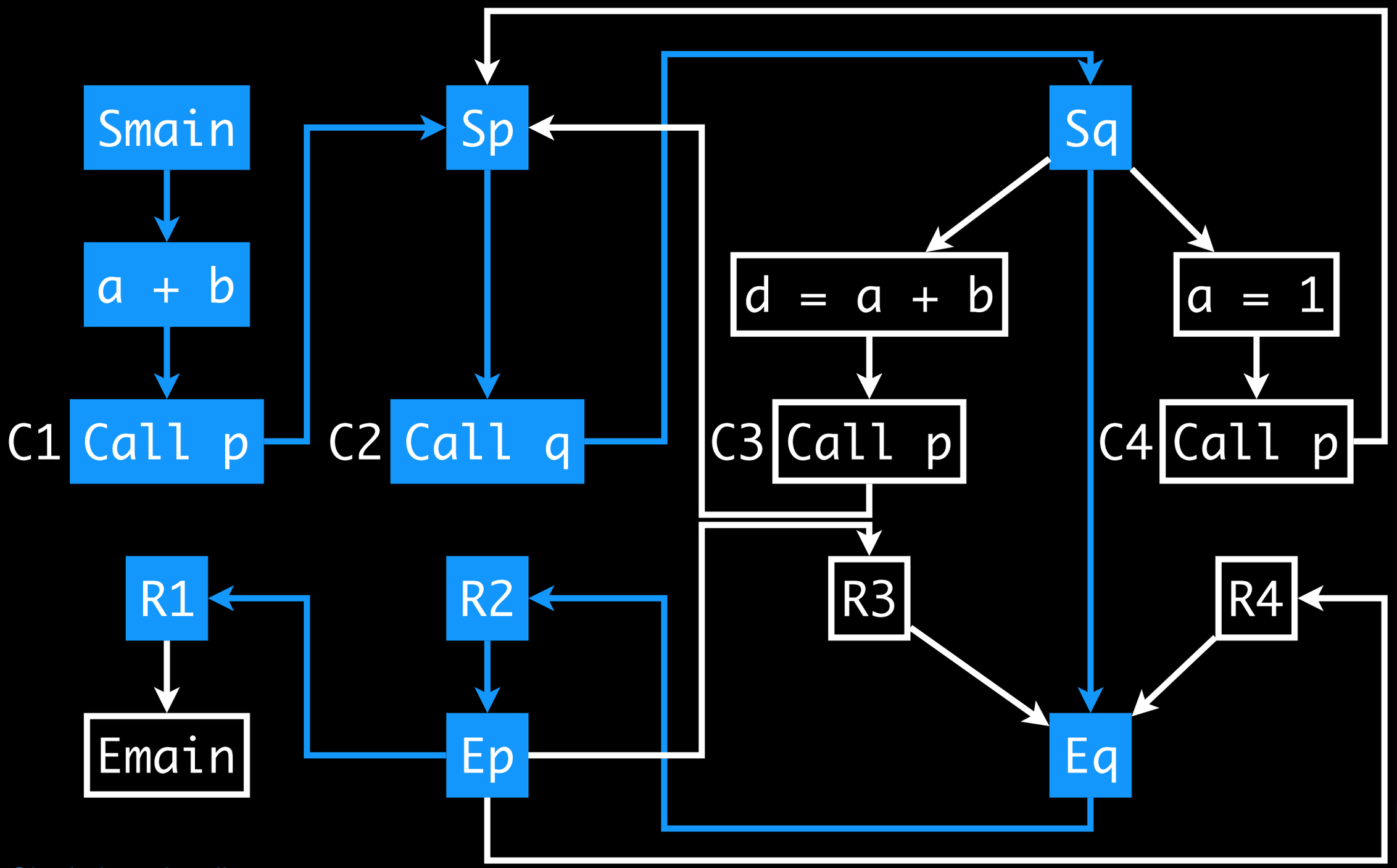
Valid/Realizable Path



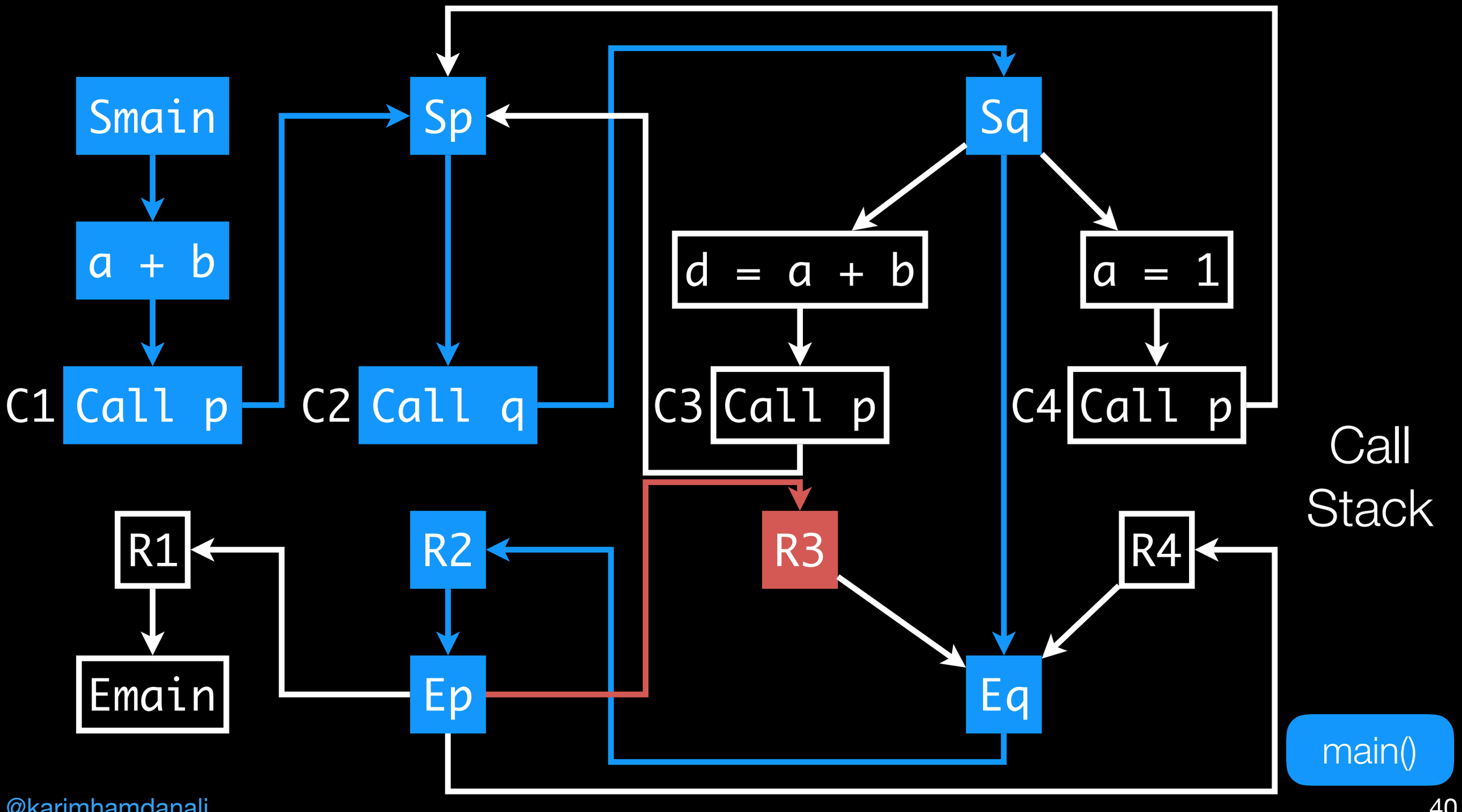
Valid/Realizable Path



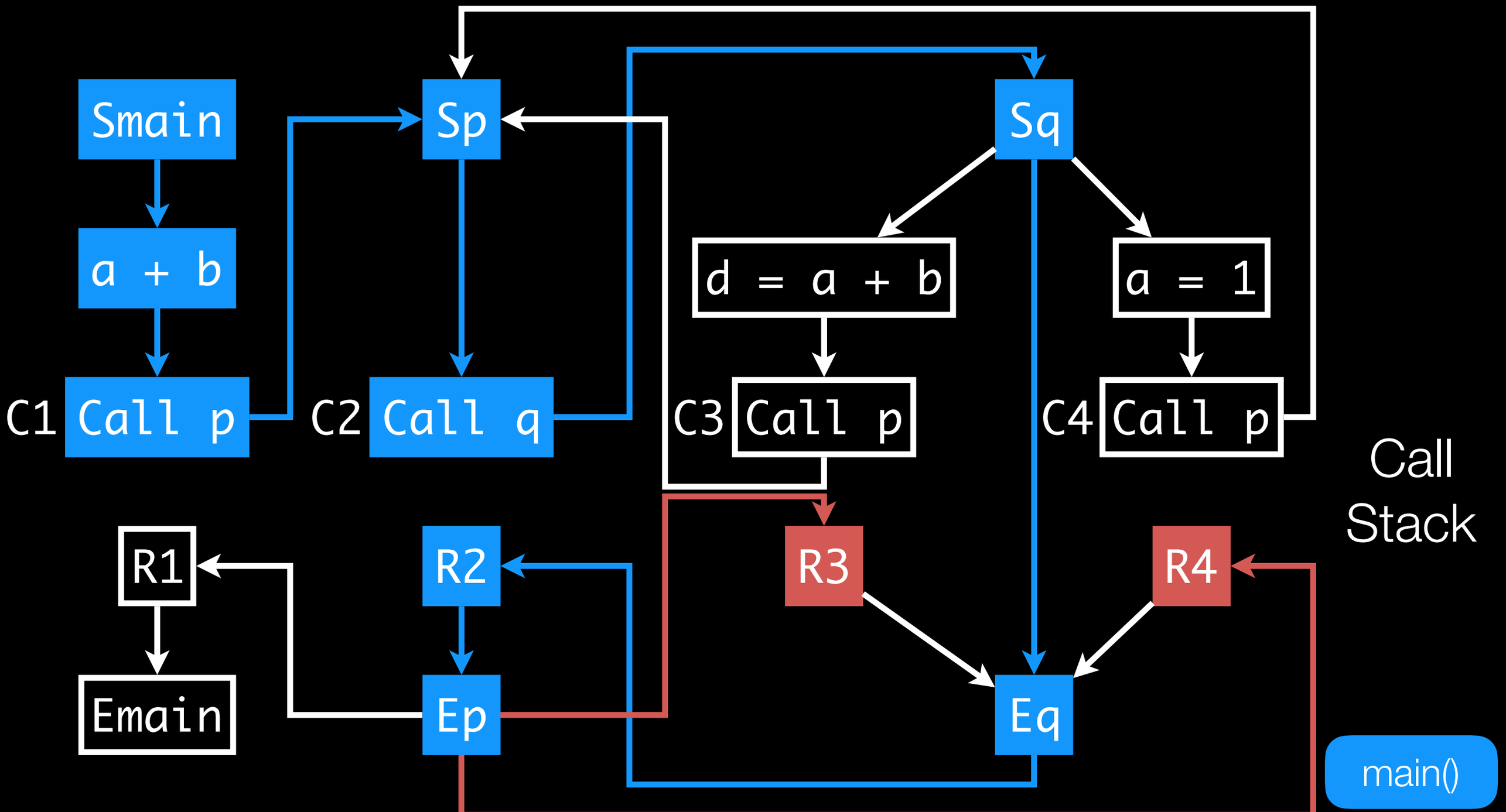
Valid/Realizable Path



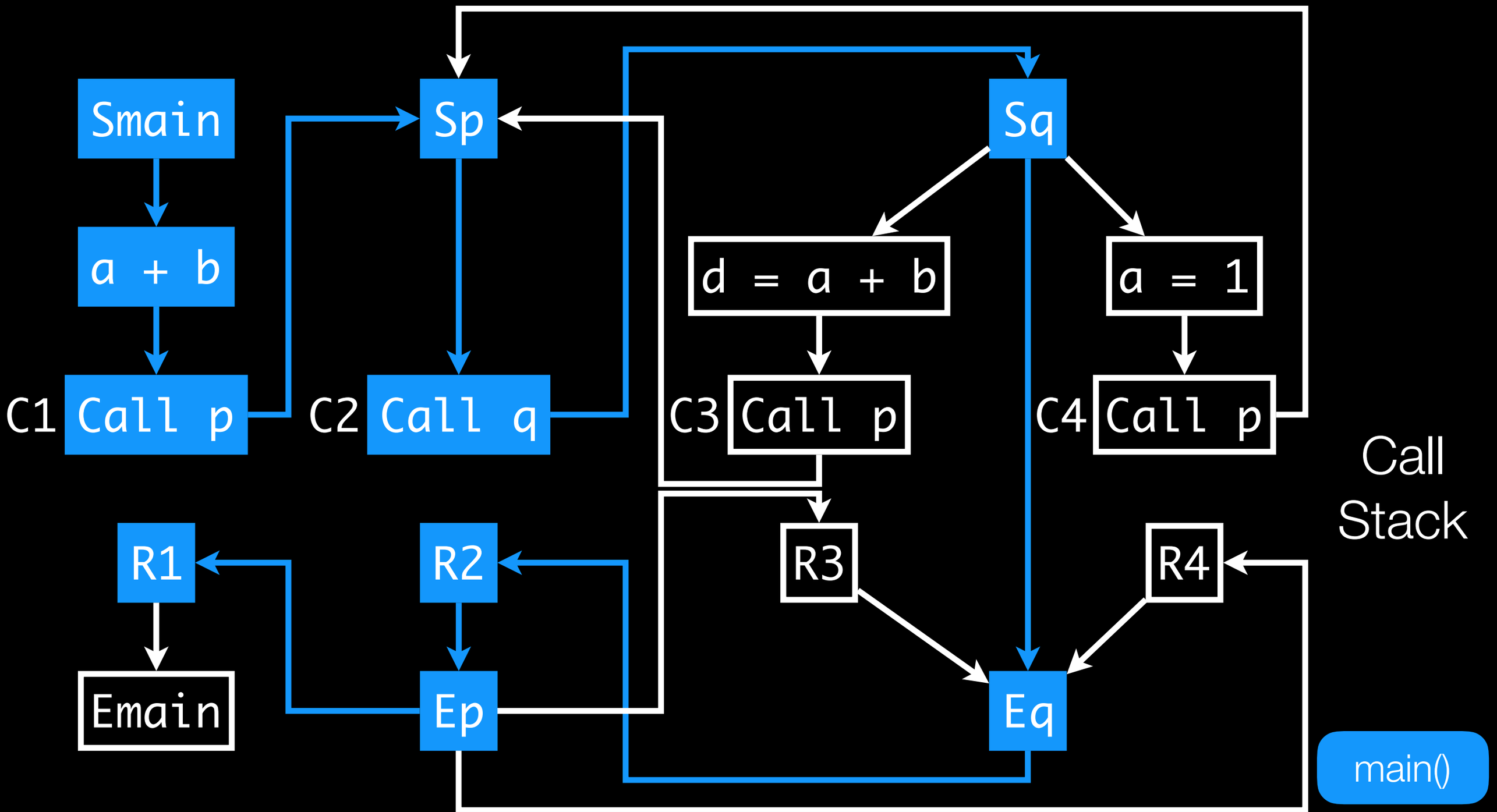
Valid/Realizable Path



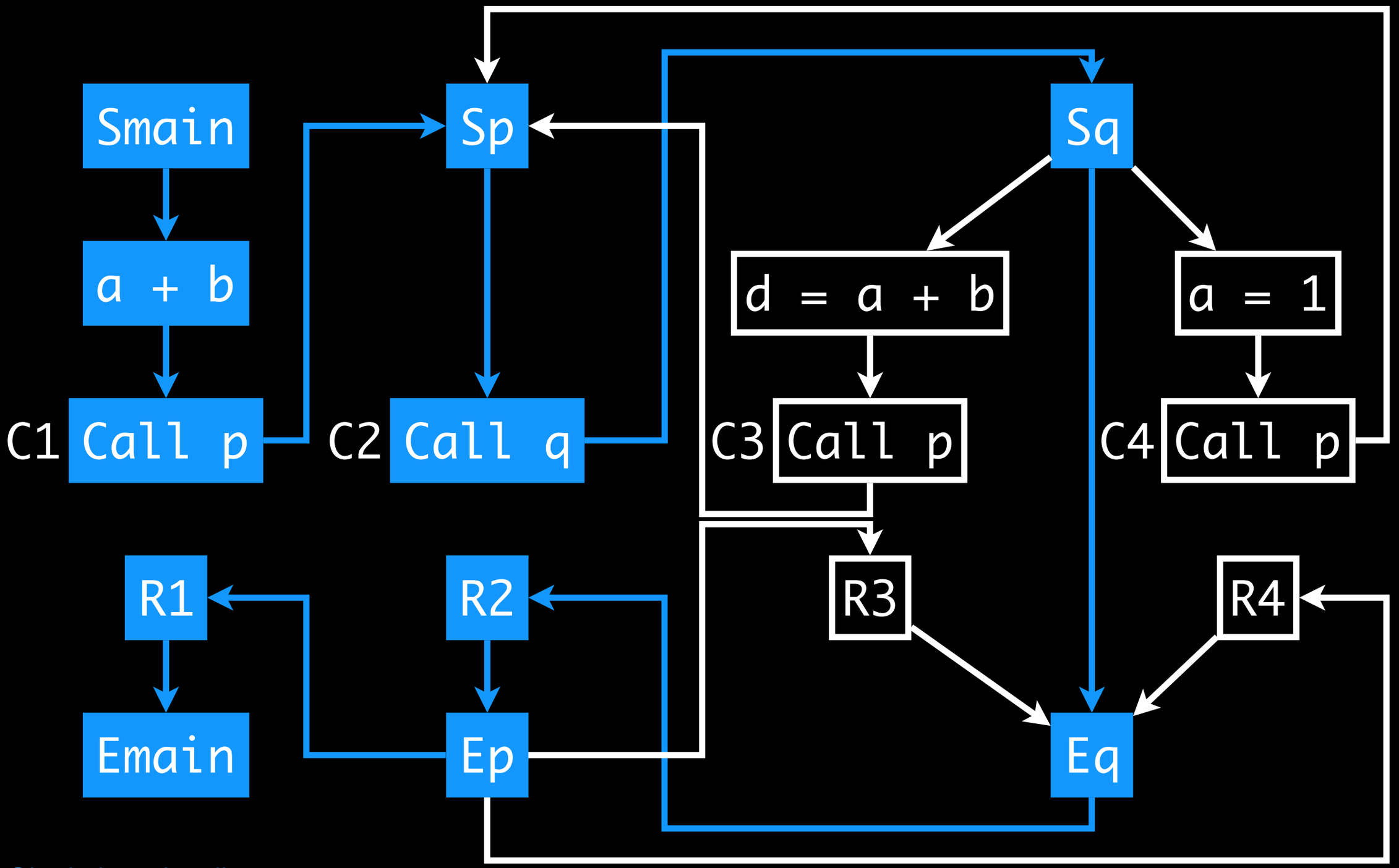
Valid/Realizable Path



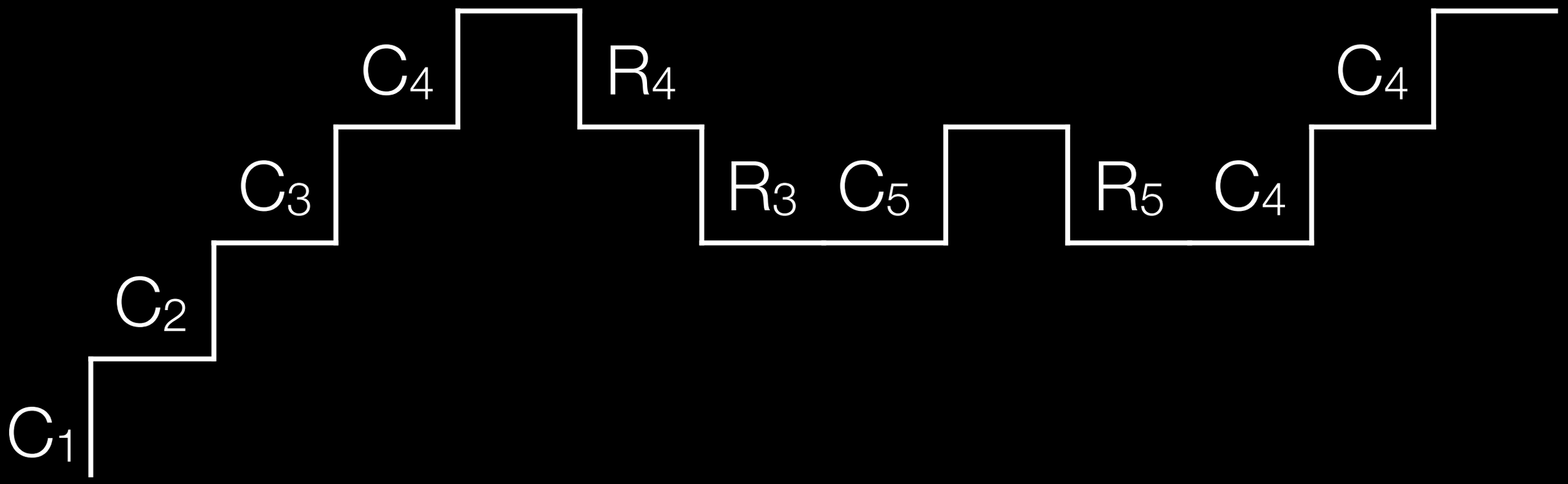
Valid/Realizable Path



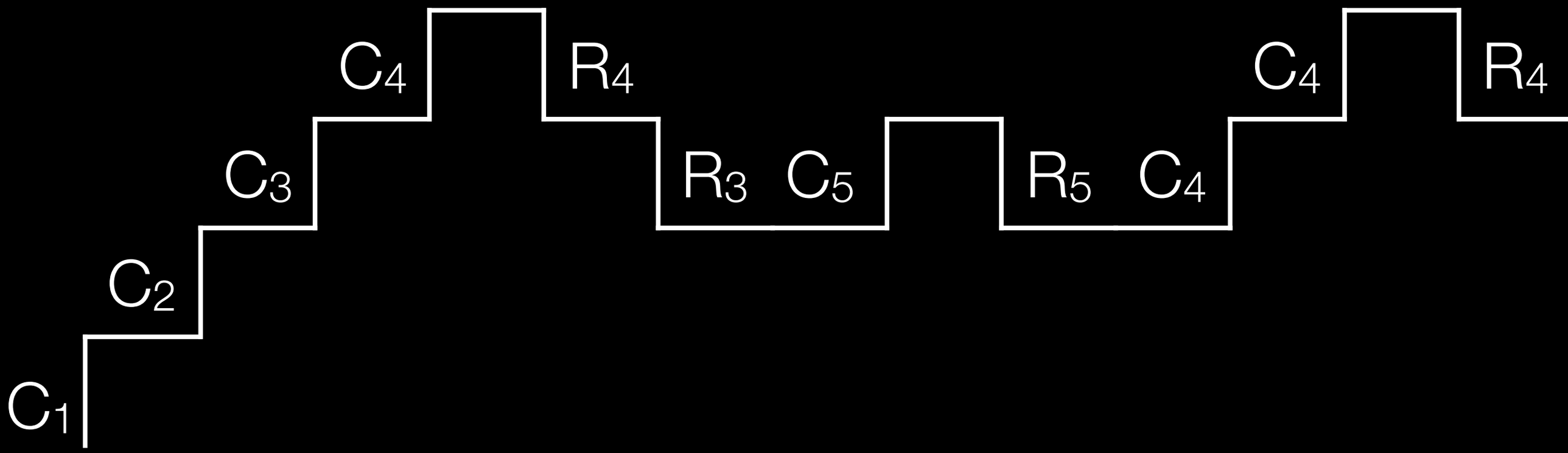
Valid/Realizable Path



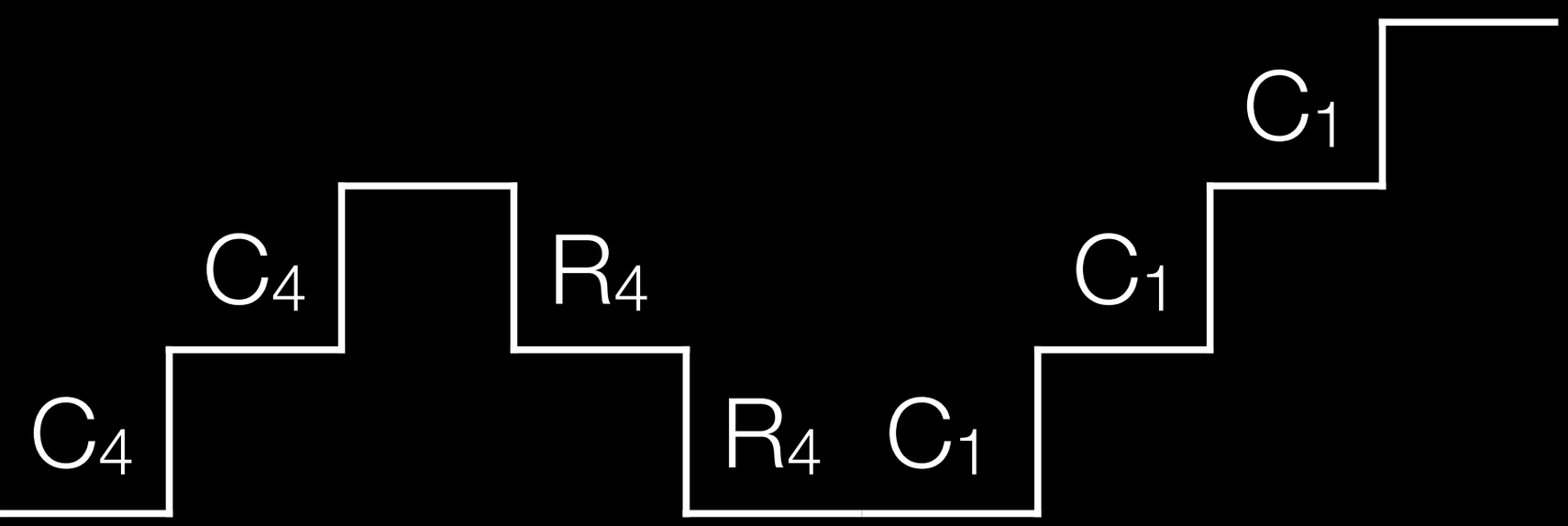
Recognizing Invalid Paths
Staircase of Calls & Returns



Recognizing Invalid Paths
Staircase of Calls & Returns



Recognizing Invalid Paths
Staircase of Calls & Returns



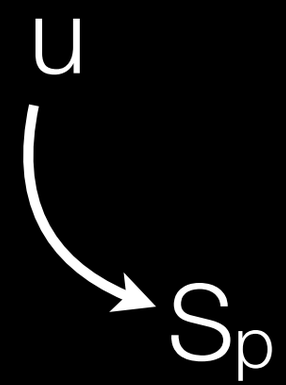
Every descending step must match
a corresponding ascending step

2 problems with that!

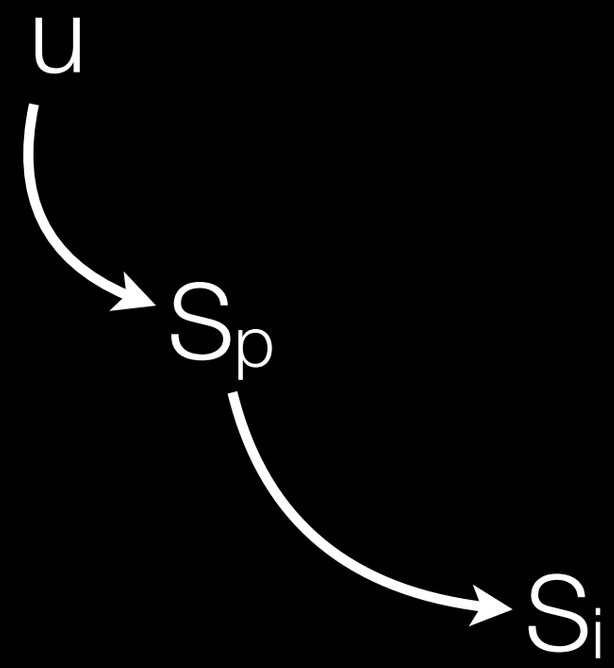
Problem # 1: Recursion

u

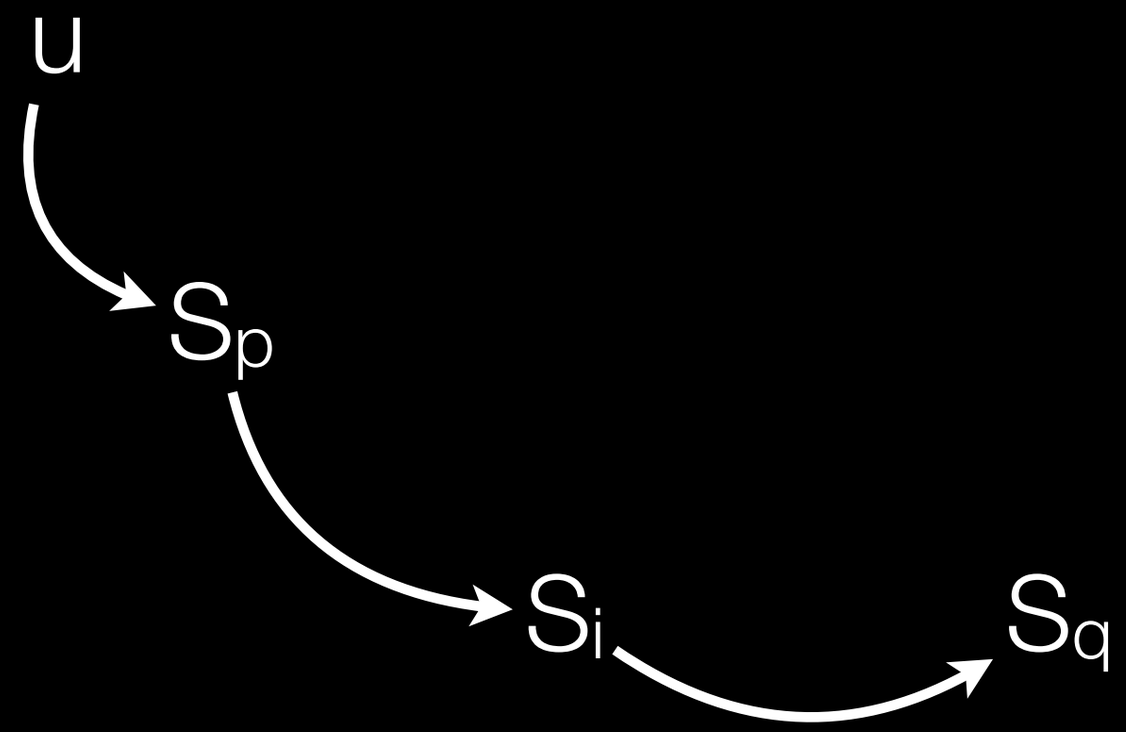
Problem # 1: Recursion



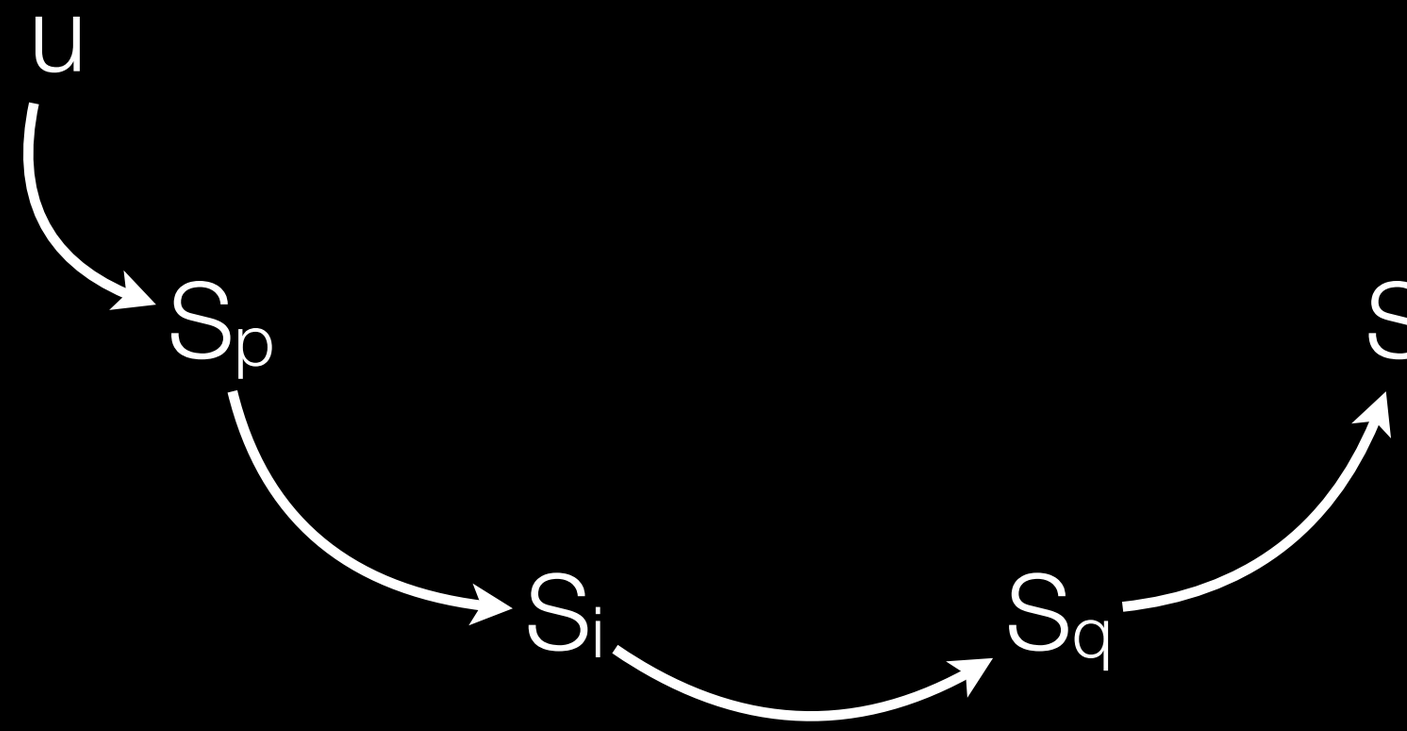
Problem # 1: Recursion



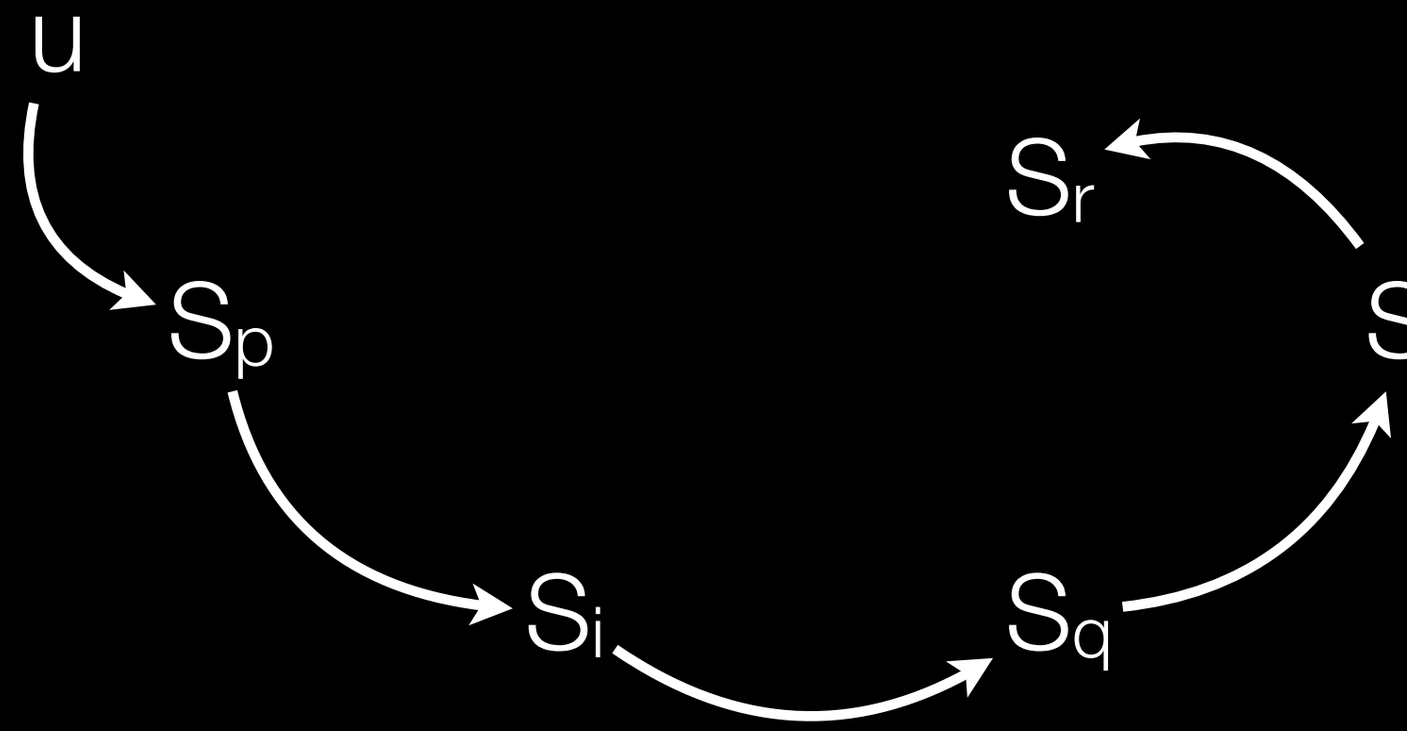
Problem # 1: Recursion



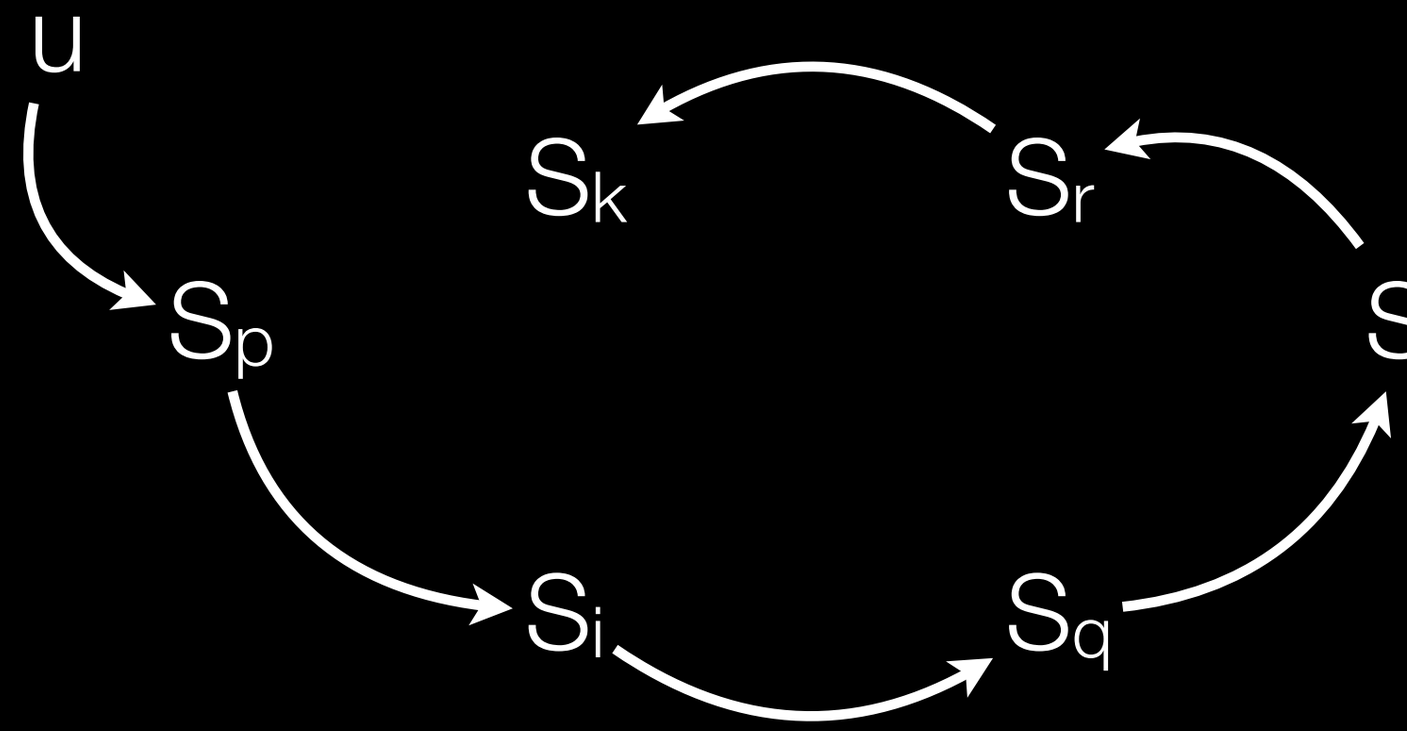
Problem # 1: Recursion



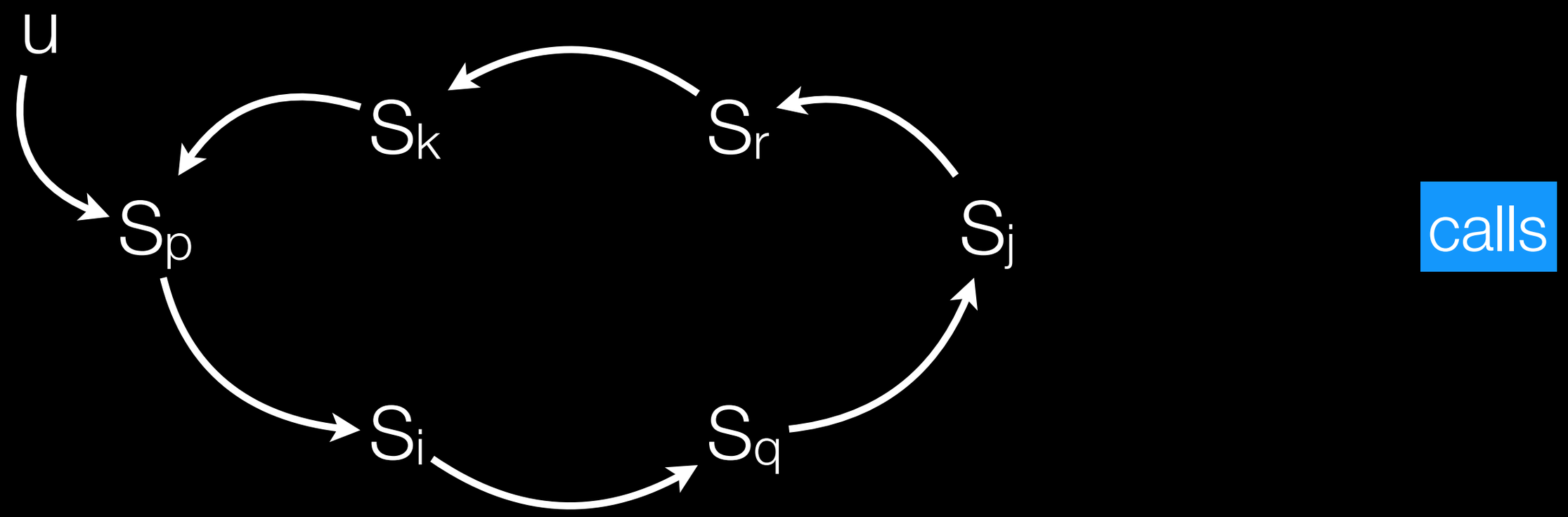
Problem # 1: Recursion



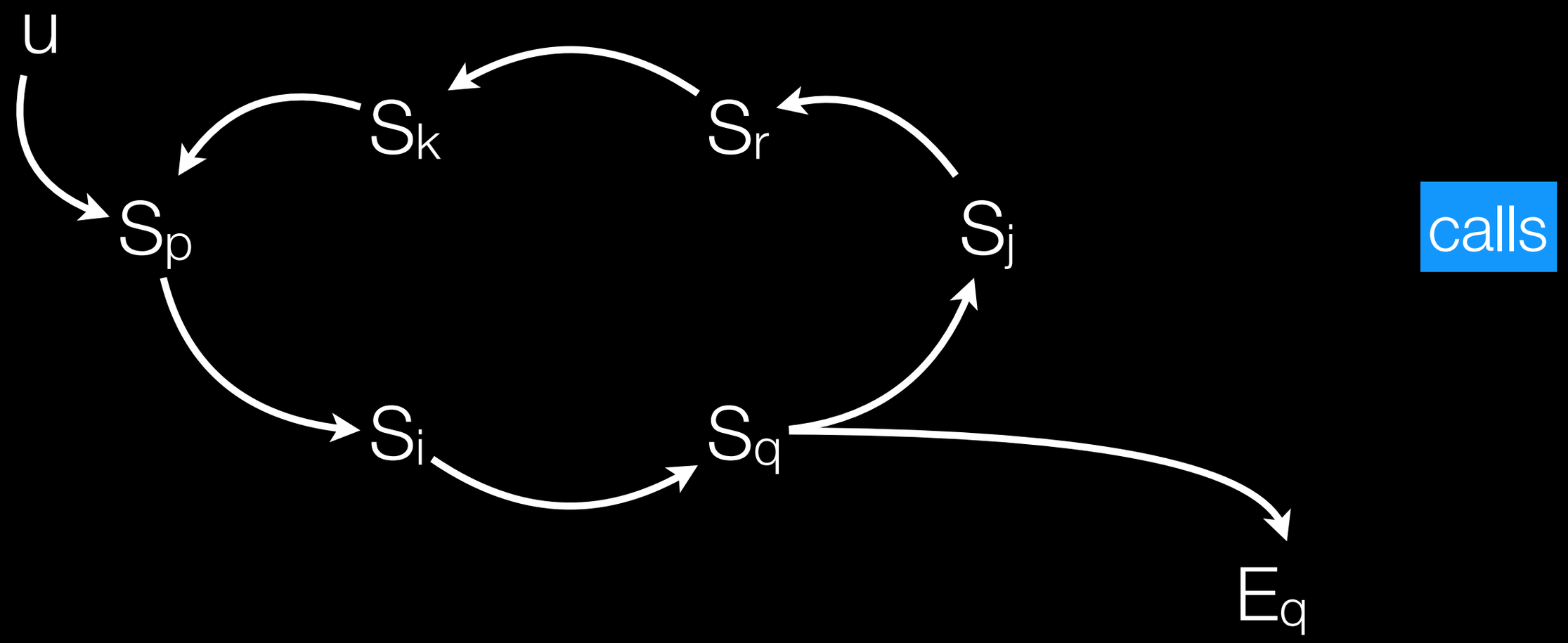
Problem # 1: Recursion



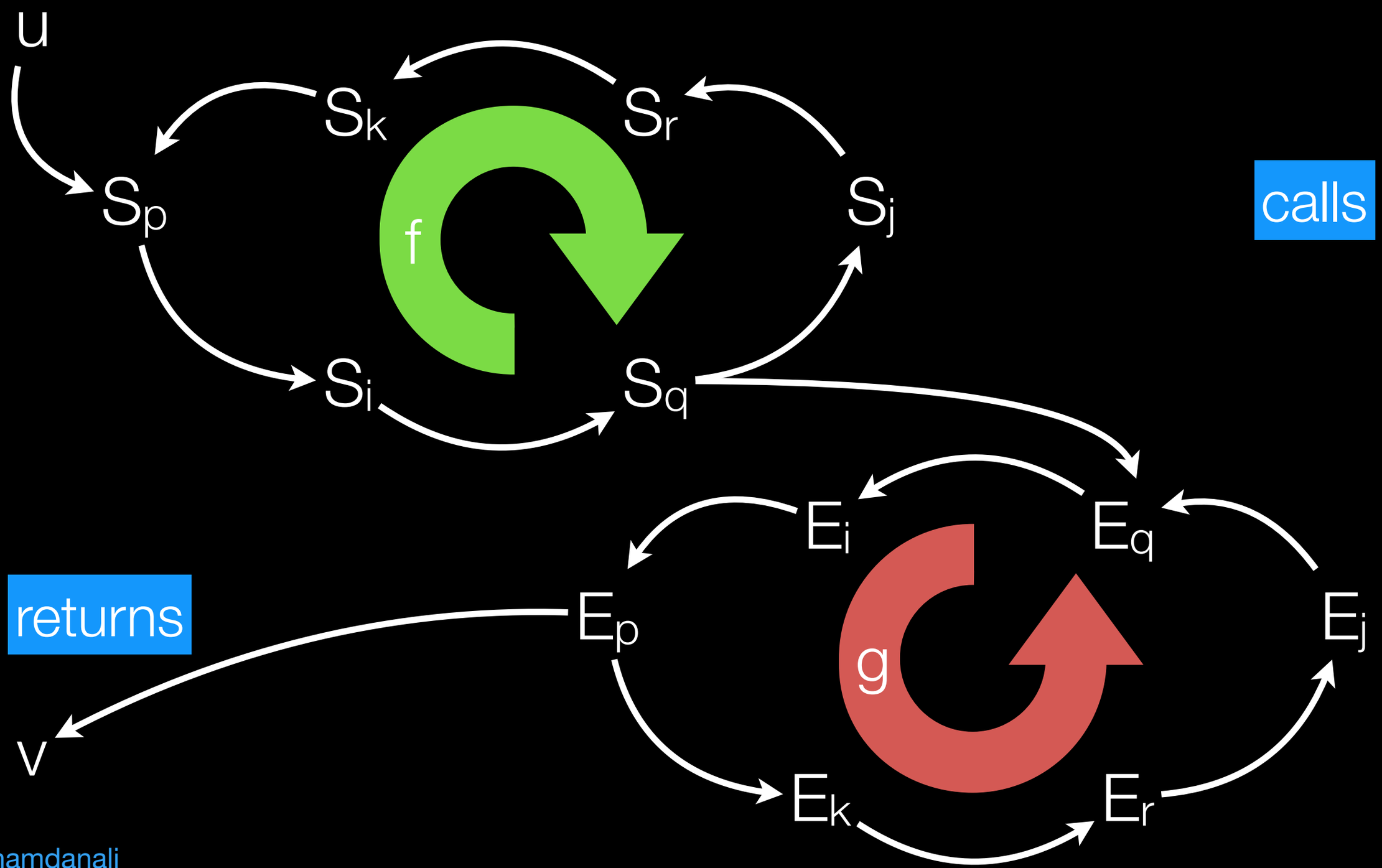
Problem # 1: Recursion



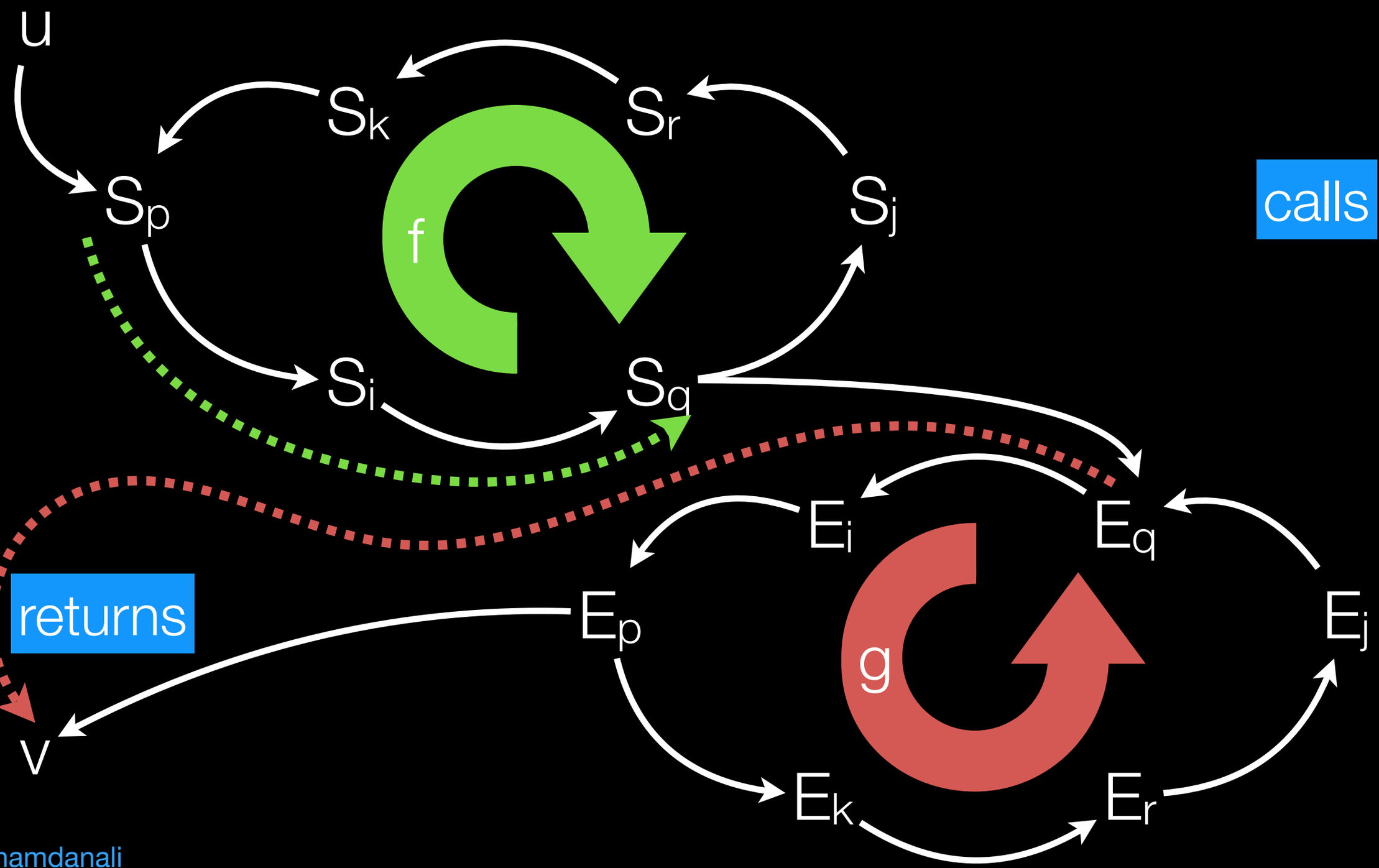
Problem # 1: Recursion



Problem # 1: Recursion



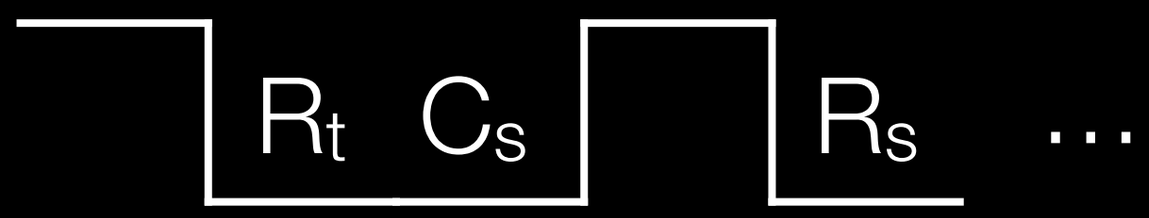
Problem # 1: Recursion



Problem # 2: Demand-Driven Analysis

```
main() {
   s = secret();
   foo(s);
   t = “123”;
   foo(t);
}
foo(v) { leak(v); }
```

assume we search from `foo(v)` backwards to find possible inputs



here: “unbalanced return” without a call must return to all possible callers

Solution:

Context-Sensitive Analysis

Context-Sensitive Analysis

- Analyze the same method, depending on the context of the current call to that method

Context-Sensitive Analysis

- Considerations:
  - How to distinguish different contexts?
  - Which contexts can be merged?

Types of Context

- A call string that encodes the methods/call sites on the current call stack
- A value context that uses the input domain values as context
- An object context that uses the currently executing object as context
- and more…

Important

Language Features

Recursion

- Must bound computation and contexts
- Often uses flow-insensitive analysis to over-approximate

Parameters/Return Values

- Must map actuals to formals and vice versa
- Don’t propagate too much info:
  - at a call: propagate only the facts relevant to that callee
  - at a return: propagate only the facts relevant to the caller
- Question: what to do with static fields?

```
main(){
   x = source();
   y = x;
   z = foo(x);
}

foo(a) {
   b = a;
   return 0;
}
```

Aliasing

aliases might be created by callers and callees

```
main(){
    a.f.g = source();
    foo(a,b);
    leak(b.f.g);
}

foo(x,y) {
    y.f=x.f;
}
```

Virtual Dispatch

- Multiple possible call targets per call site
- Consider them all!
  - “may” or “must” analysis?
  - similar to intra-procedural branches at if-then-else constructs (combine)

Threads

- Intra-procedural analyses are typically sound despite multi-threaded execution
- Inter-procedural analyses are typically *un*sound if flow-sensitive!
- Flow-insensitive analyses not impacted by multi-threading
- Effective modelling of synchronization constructs is a big open research problem!

Library Dependencies

- Typically analyze an application with its dependencies
- But what about native code?
- Often need to resort to hand-crafted summaries
- Possible way out: summarization (e.g., Averroes)

Recap

- Context-sensitivity analyzes a method multiple times, once per context
- Challenges: Recursion, parameters, aliasing, virtual dispatch, threads, libraries

Next

- Context sensitivity